## Contents

Preface ................................................................. III

Types (Chair: Ana Bove) ................................................ 1
Thomas Schilling. *Constraint-free Type Error Slicing.* ................. 1
Adam Gundry. *Type Inference for Units of Measure.* .................. 17
Julien Signoles. *An OCaml Library for Dynamic Typing.* ............ 36
Vladimir Komendantsky. *Subtyping by folding an inductive relation into a coinductive one.* ........................................ 52

Static Analysis (Chair: Henrik Nilsson) .................................. 62
Maria Christakis, Konstantinos Sagonas. *Static Detection of Deadlocks in Erlang.* .................................................. 62
Attila Góbi, Olha Shkaravska, Marko Van Eekelen. *Size Analysis of Higher-Order Functions.* ........................................ 77
Stefan Holdermans, Jurriaan Hage. *Refinement Types for Pattern-match Analysis.* .................................................. 92

Compiling (Chair: Kostis Sagonas) ....................................... 99
Jocelyn Serot, Greg Michaelson. *Compiling Hume down to gates.* ...... 130

Distributed and Parallel Programming (Chair: Kevin Hammond) ........ 146
Jean-Christophe Filliatre, K. Kalyanasundaram. *Functory: A Distributed Computing Library for Objective Caml.* .................. 146
Jost Berthold, Ken Friis Larsen, Fritz Henglein, Andrzej Filinski, Mogens Steffensen, Brian Vinter. *Functional High Performance Financial IT - The HIPERFIT Research Center in Copenhagen.* ............ 172
Parallel Programming (Chair: Viktoria Zsok) ........................................... 188
Mustafa Aswad, Phil. W. Trinder, Hans-Wolfgang Loidl. Architecture
Aware Parallel Programming in Glasgow Parallel Haskell (GPH). ... 188
Christopher Brown, Hans-Wolfgang Loidl, Kevin Hammond. Refactoring
Parallel Haskell Programs. ...................................................... 204
Silvia Clerici, Cristina Zoltán. Exploiting Parallelism by Customizing Eval-
uation. .................................................................................. 220

Data Structures (Chair: Peter Achten) ................................................. 236
Michael Lesniak. Thread-safe Priority Queues in Haskell based on Skiplists 236
Milan Straka. Adams’ Trees Revisited - Correct and Efficient Implementation. 251

Miscellaneous 1 (Chair: Marko van Eekelen) ....................................... 267
Marco T. Morazan. The Time of Space Invaders Will Come to Pass: A
Video Game Journey from Structural Recursion to Accumulative Re-
cursion in CSI. ...................................................................... 267
Jeroen Henrix, Rinus Plasmeijer, Peter Achten. GiN: a graphical language
and tool for defining iTask workflows. ..................................... 283
Pieter Koopman, Rinus Plasmeijer. Effective Test Set Generation. ..... 299

Miscellaneous 2 (Chair: Jurriaan Hage) ................................................. 309
Gergely Dévai. Restricted Function Patterns. ................................. 309
Germán Andrés Delbianco, Mauro Jaskelioff, Alberto Pardo. Applicative
Shortcut Fusion. ................................................................... 325
Christoph Herrmann, Edwin Brady, Kevin Hammond. Dependently-typed
Programming by Composition from Functional Building Blocks. .... 340
Preface

The 12th International Symposium on Trends in Functional Programming (TFP) was held in Madrid, Spain, from May 16th to May 18th, 2011. It was hosted by the Computer Science Faculty of the Universidad Complutense de Madrid, one of the oldest universities in Europe (it was founded in the XV century). In this edition, TFP was co-located with the 2nd International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA 2011), held in the same premises on May 19th. This collocation was profitable for the two symposia since a number of people registered to both events.

The TFP symposium is an international forum for researchers with interests in all aspects of functional programming, taking a broad view of current and future trends in the area of functional programming. It aspires to be a lively environment for presenting the latest research results, and other contributions, described in draft papers submitted prior to the symposium. A formal post-symposium refereeing process then selects a subset of the articles presented at the symposium and submitted for formal publication, as a Springer Lecture Notes in Computer Science volume.

The TFP symposium is the heir of the successful series of Scottish Functional Programming Workshops. Previous TFP symposia were held in Edinburgh (Scotland) in 2003, in Munich (Germany) in 2004, in Tallinn (Estonia) in 2005, in Nottingham (UK) in 2006, in New York (USA) in 2007, in Nijmegen (The Netherlands) in 2008, in Komarno (Slovakia) in 2009, and in Oklahoma (USA) in 2010.

There was a screening phase which selected works for the symposium both in the form of full papers (16 pages each), and of extended abstracts (6 to 10 pages each). However, extended abstracts were required to be extended to full papers before including them in the draft proceedings. The screening acceptance is usually based on the requirement that the work should be in scope and contain relevant information for the TFP audience. This year, 28 works were submitted and 24 of them were accepted for presentation.

TFP pays special attention to Ph.D. students, acknowledging that they play a key role in developing new trends. This attention is reflected in several ways. In the first place, there is a category of student paper to identify works that are mainly produced by students. We have accepted 9 of such papers in the screening phase. These works receive an extra round of feedback by the Program Committee before they are submitted to the standard review process for formal publication. In this way, students can upgrade their papers before they are put to compete with more ‘professional’ ones. Also, every year there is a best student paper award to acknowledge the best work made by Ph.D. students. This will be selected by the Program Committee among the student papers accepted for formal publication. To promote the participation of students, this year the TFP 2011 organization has offered a number of grants in order that students could get a half-price fee. We have awarded 12 of these grants.

In this edition, 46 participants registered to the symposium, coming from 11 different countries. The scientific program included an invited talk by Neil Mitchell (UK), an expert in transformation and analysis of functional Programs. The title of his talk was Finding functions from types. The social program consisted of a visit to the beautiful city of Toledo, and a conference dinner celebrated there. This city is located 70 Km south-west of Madrid, and it is known as the city of the three cultures, because the centuries-long coexistence of Christian, Jewish and Muslim cultures in the Middle Age.
Acknowledgements

We thank all the speakers, the authors, and the rest of participants for contributing to the success of TFP 2011. We also thank the Program Committee for preparing such a exciting program, the local Organizing Committee for all the arrangements which have made possible this event, and the TFP Steering Committee for providing advice and support. Finally, we acknowledge the generous funding of our sponsors the Spanish Ministry of Science and Innovation, and the Computer Science Faculty of the Universidad Complutense, and the administrative support given for the Fundación General de la Universidad Complutense along all the process.

Ricardo Peña
TFP 2011 Program Committee Chair
TFP 2011 Organizing Committee Chair

Marko van Eekelen
TFP Steering Committee Chair

Madrid, May 2011
Program Committee

Peter Achten  Radboud University Nijmegen, NL
Ana Bove  Chalmers University of Technology, SE
Olaf Chitil  University of Kent, UK
Marko van Eekelen  Radboud University Nijmegen and Open University, NL
Robert B. Findler  Northwestern University, USA
Víctor Gulías  University of La Coruña, ES
Jurriaan Hage  University of Utrecht, NL
Kevin Hammond  University of St. Andrews, UK
Michael Hanus  Christian Albrechts University of Kiel, DE
Zoltán Horváth  Eötvös Loránd University, HU
Frank Huch  Christian Albrechts University of Kiel, DE
Mauro Jaskelioff  National University of Rosario, AR
Rita Loogen  Philipps University Marburg, DE
Jay McCarthy  Brigham Young University, USA
Henrik Nilsson  University of Nottingham, UK
Rex Page  University of Oklahoma, USA
Ricardo Peña (PC Chair)  Complutense University of Madrid, ES
John Reppy  University of Chicago, USA
Konstantinos Sagonas  Uppsala University, SE
Simon Thompson  University of Kent, UK
Germán Vidal  Polytechnical University of Valencia, ES

Organizing Committee

Steering Committee Chair:
Marko van Eekelen  Radboud University Nijmegen and Open University, NL

Steering Committee Treasurer:
Hans-Wolfgang Loidl  Heriot-Watt University, UK

Steering Committee Secretary:
Marco T. Morazán  Seton Hall University, USA

Symposium Organization Chair:
Ricardo Peña  Complutense University of Madrid, ES

Local Organizing Committee

Puri Arenas  Complutense University of Madrid, ES
Clara Segura  Complutense University of Madrid, ES
Enrique Martín Martín  Complutense University of Madrid, ES
Juan Rodríguez Hortalá  Complutense University of Madrid, ES
Manuel Montenegro  Complutense University of Madrid, ES
Abstract. Type error messages for ML-based languages tend to suffer from imprecise error locations – the type checker reports only one of many possible locations of an error. The notion of a type error slice corrects this by reporting all program locations that contribute to a given error (and no more). Previous work on producing type error slices required the use of a constraint-based type checker implementation. For most existing systems this would require substantial changes to well-tested and subtle pieces of code. In this work we show how to produce useful type error slices with an unmodified type checker. Other tools, such as automatic correction systems, can be layered on top of our system. We have implemented this technique on top of the Glasgow Haskell Compiler (GHC) and report our experiences.

1 Introduction

Consider the following ill-typed Haskell definition:

\[ f = \lambda x \rightarrow \text{length} (x \ 'a' \ ++ \ x \ [\text{True}]) \]

The widely used Glasgow Haskell Compiler\(^1\) (GHC) reports the error:

**Couldn’t match expected type ‘Char’ with actual type ‘[t0]’**

The associated location is the application \((x \ [\text{True}])\). GHC somehow decided that the “actual type” of \(x\) should be \(\text{Char} \rightarrow [t]\) (for some \(t\)). If we swap the two calls of \(x\), however, GHC changes its mind and decides that \(\text{Bool} \rightarrow [t]\) is the “actual type”.

\[ f = \lambda x \rightarrow \text{length} (x \ [\text{True}] \ ++ \ 'a') \]

**Couldn’t match expected type ‘[Bool]’ with actual type ‘Char’**

While one might argue that the wording of the message could be improved, the real problem is more fundamental. GHC only reports one location for the error. In a language based on the Hindley/Milner typing discipline [6, 1], however,

\(^1\) Version 7.0.2
it is impossible to always report the correct location of an error; there are in general several possible locations to fix an error, depending on the intended semantics of the program.

A better notion of location for error messages in such languages are type error slices [2]: the location of an error includes all parts of the program that contribute to it (and no more). For the first example above this would be:

\[
f = \lambda x \rightarrow \text{length} \left( x \ [\text{True}] \ + \ x \ '*' \right)
\]

All the highlighted parts of the program together cause the error, and only changes to these locations may remove the error. In particular, no changes to other parts of the program can possibly fix the error. It is also possible to provide a textual representation of error slices:

\[
\ldots (\lambda x \rightarrow \ldots (x \ [\ldots]) \ldots (x \ '%') \ldots) \ldots
\]

From the above type error slice (or just error slice for short) we can deduce quite a bit of useful information about the error. First, it is important that \( x \) is \( \lambda \)-bound and therefore has a monomorphic type. Second, it is used as a function in two different applications. Third, the fact that the literal \texttt{True} is not highlighted indicates that it is not important that \( x \) is applied to a list of booleans, but merely that \( x \) is applied to some list.

Type error slices are not a replacement for good error messages, but they can provide helpful complementary information.

Type error slices for a language with polymorphic types were first introduced by Haack and Wells [2] and their construction was described for a small ML-like language. Rahli et al. [8] later implemented a type error slicer for Standard ML. Stuckey et al. [10, 12] use type error slices as the basis for more traditional error messages. Their system, Chameleon, would for example also display inferred types for each use site of the variable \( x \) in the examples above.

All these systems are based on a constraint-based implementation of the type checker. The source program is parsed and then translated into constraints expressing the typing rules. If the source program had a type error then these constraints are unsolvable and it is then possible to find a minimal unsolvable constraint set. These are the constraints that are in a sense essential to the error. Because it is minimal, removing any constraint from this set makes it solvable. Finally, this set is translated back into source code locations which then form the type error slice. The solid arrows in Figure 1 outline this process.

While it is known that type-checking for Hindley-Milner corresponds to constraint-solving, most implementations generate and solve these constraints on the fly. This is insufficient for the way type error slices are constructed in the aforementioned systems. For example, Rahli et al. [8] re-implemented a full type checker for SML. Type checkers are complicated pieces of code and are hard to get right. Having to maintain two implementations is usually unacceptable.

\[2\] Well, we could insert a shadowing binding of \( x \), but this means actually changing the use sites of \( x \) since they now refer to a different variable.
Even type checkers that \emph{are} implemented in terms of explicit constraint-generation are not immediately suitable for the construction of type error slices. In order to construct the type error slice, it is necessary to accurately associate source code locations with the constraints they introduced.

In this work we present a simple method of constructing accurate type error slices without special requirements on the type checker. Instead of translating the source program into a set of constraints, our method works directly on the source program. Our approach is based on an idea from the \textsc{Seminal} project [5, 4]: in order to check whether a program location is part of the type error slice, we modify the program in a way that removes any type constraints introduced by that location. If the modified program type checks (or uncovers a different error), the location must be part of the type error slice.

Coming back to our first example, to test whether the first use site of \(x\) should be part of the error slice, we replace \(x\) with \(\bot\), that is, a term that always type checks. A possible such term in Haskell is \((\text{let } y = y \text{ in } y)\), but any term of type \(\forall \alpha. \alpha\) will do. We obtain the following program which is sent to the type checker:

\[
\text{f} = \lambda x \rightarrow \text{length} (\bot [\text{True}] ++ x \;'*)
\]

This program does indeed type check, so we conclude that the first use site of \(x\) must be part of the error slice. To test whether the call to \text{length} takes part in the error we send this program to the type checker:

\[
f = \lambda x \rightarrow \bot (x [\text{True}] ++ x \;'*)
\]

This program does not type check, which means that “disabling” the reference to \text{length} did not fix the error, and thus it is not part of the error slice. To test whether the application of \(x\) to \([\text{True}]\) contributes to the error, \((x [\text{True}])\) is rewritten to \(\bot x [\text{True}]\). That is, we keep both function and argument, but we remove all constraints that were introduced by applying one to the other. We obtain the program:

\[
f = \lambda x \rightarrow \bot (\bot x [\text{True}]) (x \;'*)
\]

The resulting program now type checks, hence this application site must be part of the type error slice.
Our type error slicer systematically performs such rewriting steps for every location of the input program until a minimal set of program locations is found which are then reported as the type error slice. Eventually we arrive at the following program, which corresponds to the expected error slice.

\[
\bot \ (\lambda x \rightarrow \bot \ (\bot \ x \ [\bot]) \ (x \ '*)')
\]

Because this technique is purely source-based it can also quite easily be adapted to support some extensions to the Hindley/Milner type system.

This work makes the following contributions:

- We show how constraint-based type error slicing can be implemented while treating the type checker as a complete black box (Section 5). This is predicated upon the existence of a source-level rewriting function that simulates removal of type checker constraints.
- As a concrete example we define such a function for Mini-ML (Section 5). (Mini-ML is defined in Section 3.)
- We describe our findings after implementing this approach for Haskell.

The remainder of the paper is organised as follows. Section 2 describes the idea of type type error slices in more detail. We then introduce the formal framework to accurately describe our technique. For this, Section 3 defines the syntax and constraint-generation rules for the small ML-like language Mini-ML. Section 4 reviews how to generate type error slices from the generated constraints, and Section 5 then explains our novel method of achieving the same result using an existing type checker and source-level rewritings. In Section 6 we outline our findings of applying this method to Haskell. Section 7 then lists future work and Section 8 concludes this paper.

## 2 Type Error Slices

The Hindley/Milner type system\(^\text{3}\) [6] has the remarkable feature of requiring no type annotations from the user while still supporting polymorphic types. The downside is that it makes it difficult to give helpful error messages because assumptions the programmer made about the code are not explicit.\(^\text{4}\) Instead, the types of sub-expressions are inferred by the type checker and checked for consistency. As a result, the point at which an error is discovered is highly dependent on the algorithm used for type checking.

Many researchers have worked on techniques to control the location of the reported error or to find the most likely source of an error and report this together with a helpful error message. A good survey is given by Heeren [3].

\(^\text{3}\) Also known as ML type system or Damas/Milner type system.

\(^\text{4}\) It is often considered good style to add type signatures to top-level functions, but this is not required. Additionally, local definitions typically do not contain type annotations either.
to work well for commonly made errors, for instance by beginning programmers. However, as programmers become more experienced, more complicated errors may no longer match the assumptions made by the error locating heuristics. The result is a misleading error message which blames the wrong part of the program.

Type error slices provide an unbiased notion of an error’s location. Instead of only a single location (decided by heuristics or implementation details of the type checker) a type error slice consists of all of the locations that together cause the error. This means in particular that any of the program parts in the type error slice may be changed to resolve the error. Changes to any other parts of the program cannot fix the error.

Consider the following ill-typed program.

```haskell
split [] = ([], [])
split [x] = ([], x)
split (x:y:zs) = 
  let (xs, ys) = split zs in (x:xs, y:ys)
```

The programmer probably meant to return the singleton list `[x]` instead of just `x` as the second result in the second case alternative. However, the type checker cannot know that; it detects the first inconsistency when trying to unify the inferred type of the first two cases, `[[a]] → ([b], [a])`, with the type inferred for the last case, `[t] → ([t], [t])`. This leads to the equality `t ≡ [a] ≡ a`, which is not allowed in Haskell. GHC reports this error with the following message, blaming the last occurrence of `ys`.

```
Occurs check: cannot construct the infinite type:
a = [a]
  Expected type: [[a]]
  Inferred type: [a]
In the second argument of ‘(:)’, namely ‘ys’
In the expression: y : ys
```

A type error slice for the same error with all the relevant locations highlighted looks as follows.

```haskell
split [] = ([], [])
split [x] = ([], x)
split (x:y:zs) = 
  let (xs, ys) = split zs in (x:xs, y:ys)
```

All the highlighted source code fragments are necessary for the error to occur. There are, in fact, several possible type error slices for this error. We will come back to this issue in Section 4.1.

The highlighted locations tell us several useful things about the error:
- The first equation is not relevant to the error since it does not contain any highlighted locations.
- Only the second component of the returned tuple plays a rôle in the error.
- The recursive call to \texttt{split} is not necessary for the error.
- The \texttt{ys} part of the expression \texttt{y:ys} is not strictly necessary for the error.

The problem in this slice is only that \texttt{y} appears as the first argument to the \texttt{cons} operator (\texttt{:}).

This shows that there are clearly multiple ways to fix the error. Which one is the correct change depends on the intended purpose of the code and is therefore impossible to predict. For example, the above error can be fixed by replacing \texttt{y:ys} with \texttt{y++ys}. This would give \texttt{split} the type \([a] \rightarrow ([a], [a])\) which may or may not be what the programmer intended.

As an aside, the usual technique of adding a type annotation with the intended type of \texttt{split} will not necessarily narrow down the exact location of the error, because the type annotation may be wrong and will therefore simply become part of the error slice! We are considering some techniques to allow the user to declare a type signature as trusted when examining an error, but this is outside the scope of this paper.

### 2.1 Presentation of Type Error Slices

There are different ways to present type error slices. Above we highlighted the slice directly in the source, but sometimes a simple, text-based representation is useful. This can be done by simply replacing the elided parts with “..”. Recall the error slice from the introduction:

\[
\lambda x \rightarrow \ldots(x ~[\ldots]) \ldots (x'\star') \ldots
\]

Note that some extra parentheses were introduced. This is due to the following reason: In many functional languages (including Haskell) function application does not have a special syntactic form; the function and its arguments are separated simply by a space. If only the application node is part of the error slice (but the exact argument may not) then the textual representation uses explicit parenthesis for the application node and angle brackets around omitted parts:

\[
\ldots(f ~(\ldots) ~4) \ldots
\]

In this slice the first argument is not part of the error slice but the second is. When highlight a slice in source code we can make this clear by creating a “framed hole”:

\[
\begin{array}{c}
f \quad \text{irrelevant} \quad 4
\end{array}
\]

Type error slices created by our algorithm and those described in the literature are always \textit{minimal}. This means that removing any of one of the locations of a slice will fix the error (or at least uncover another error). For many errors, however, there are multiple such minimal sets of locations, hence the error has
multiple error slices. Often these slices will have many locations in common, which are then more likely to fix the error. Finding all slices is not possible in general (see Section 4.1) but if more than one error slice has been found their source locations may be highlighted with different intensity (or hue) that indicates how many slices they are part of. More intensely highlighted locations then indicate more likely sources of an error.

2.2 Limitations of Type Error Slices

Not every error is suitably presented as a type error slice. This includes syntax errors or undefined variable errors, but also errors involving ambiguity. Removing constraints cannot remove ambiguity, in fact, it might create even more!

Without an error message, type error slices alone are also seldom more helpful than error messages reported by current compilers, although they can be quite helpful at excluding possible sources of an error. However, type error slices can complement error messages and may serve as the basis for more helpful error messages as demonstrated by the Chameleon system [12].

3 Constraint-based Type Checking

We start off by reviewing the constraint-based approach of type checking. Figure 2 defines the syntax of Mini-ML, a simple functional language with integers and polymorphic let-bindings. Expressions in Mini-ML are annotated with unique labels $l$ which identify each sub-expression. In a real implementation this may be source location information from the parser. The typing rules for Mini-ML are completely standard Hindley/Milner rules and they are shown in Figure 3.

A constraint-based type checker translates a source program into a set of constraints, which are then solved in a separate step. If the constraints are solvable, then the program type checks. If they are unsolvable, the program has a type error.

Figure 4 defines rules for generating equality constraints for Mini-ML (adapted from Haack and Wells [2]). These can then be solved using standard unification. The translation returns three parts:

- $\tau$ is the type assigned to the input expression.
- $C$ is a set of equality constraints. Each equality constraint is annotated with a label.
- $\Gamma$ is a bottom-up environment. It maps each free variable in the input expression to a set of type variables, one per use site.

---

5 This is the same language that Haack and Wells [2] used to introduce type error slices for Hindley/Milner.
6 It is called “bottom-up” because information travels from the leaves of the syntax tree to the root, i.e., from variables’ use sites to their binders.
\[
\text{expr ::= } \bot \quad \text{Bottom} \\
| \ x^l \quad \text{Variable} \\
| \ n^l \quad \text{Integer literal} \\
| \ (\text{expr } + \text{expr})^l \quad \text{Integer addition} \\
| \ (\lambda x^l \rightarrow \text{expr})^l \quad \text{Abstraction} \\
| \ (\text{expr } \text{expr})^l \quad \text{Application} \\
| \ (\text{let } x^l = \text{expr } \text{in } \text{expr})^l \quad \text{Non-recursive let}
\]

\[
\tau ::= \alpha \quad \text{Type variable} \\
| \ \text{Int} \quad \text{Integer type} \\
| \ \tau \rightarrow \tau \quad \text{Function type}
\]

**Fig. 2.** Expression and type syntax for Mini-ML. Expressions are annotated with unique labels \( l \).

\[
\begin{array}{c}
\Gamma \vdash \text{expr} : \tau \\
\hline
\Gamma \vdash n : \text{Int} & \text{LIT} & \alpha \text{ fresh} \quad \Gamma \vdash \bot : \alpha & \text{BOT} & \Gamma(x) = \forall \alpha.\tau & \text{fresh} \quad \Gamma \vdash x : \tau[\beta/\alpha] & \text{VAR} \\
\hline
\Gamma \vdash e_1 : \text{Int} & \Gamma \vdash e_2 : \text{Int} & \text{ADD} & \Gamma, x : \alpha \vdash e : \tau & \alpha \text{ fresh} \quad \Gamma : \lambda x \rightarrow e : \alpha \rightarrow \tau & \text{ABS} \\
\hline
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 & \Gamma \vdash e_2 : \tau_1 & \Gamma \vdash e_1 \ e_2 : \tau_2 & \text{APP} & \Gamma, x : \forall \alpha.\tau_1 \vdash e_2 : \tau_2 & \overline{\alpha} = \text{fv}(\tau_1) - \text{fv}(\Gamma) \\
\hline
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 & \text{LET} & \Gamma \vdash e_1 : \tau_1 & \Gamma, x : \forall \alpha.\tau_1 \vdash e_2 : \tau_2
\end{array}
\]

**Fig. 3.** Syntax-directed typing rules for Mini-ML. Each expression is assigned its most general type.

The rules in Figure 4 may look a bit dense, but they all follow quite simple patterns. We use rule \( \text{C-App} \) to describe the basic ideas. The first step is to generate constraints for both the expression in the function position \( e_1 \) and the expression in the argument position \( e_2 \). The application site now creates three additional constraints: (1) the type of \( e_1 \) must be of function shape, (2) the argument type of this function type must match the type of \( e_2 \), and (3) the result type of the application expression matches the result type of \( e_1 \). These three constraints are returned together with the combined constraints of both \( e_1 \) and \( e_2 \), as well as their combined environments. In fact, every constraint-based typing rule returns all the constraints of its sub-expressions plus the additional constraints imposed by the given syntactic construct.

There is a certain degree of freedom in the design of such constraint-based typing rules. For example, all the constraints added by rule \( \text{C-App} \) have the


\[
\begin{align*}
\text{expr} \downarrow \tau, C, \Gamma \\
\beta \text{ fresh} & \quad \text{C-LIT} & \beta \text{ fresh} & \quad \text{C-BOT} \\
\tau, C, \Gamma \downarrow l, \beta, \emptyset \\
\beta, \beta_x \text{ fresh} & \quad \text{C-VAR} \\
x_l \downarrow \beta, \{\beta \equiv \text{Int}\}, \{x : \{\beta_x\}\} \\
\text{e}_1 \downarrow \tau_1, C_1, \Gamma_1 & \quad \text{e}_2 \downarrow \tau_2, C_2, \Gamma_2 & \beta \text{ fresh} \\
C_0 = \{\tau_1 \equiv \text{Int}, \tau_2 \equiv \text{Int}, \beta \equiv \text{Int}\} & \quad \text{C-ADD} \\
(e_1 + e_2)l \downarrow \beta, C_0 \cup C_1 \cup C_2, \Gamma_1 \cup \Gamma_2 \\
\text{e}_1 \downarrow \tau_1, C_1, \Gamma_1 & \quad \text{e}_2 \downarrow \tau_2, C_2, \Gamma_2 & \beta, \beta_1, \beta_2 \text{ fresh} \\
C_0 = \{\tau_1 \equiv \beta_1 \rightarrow \beta_2, \tau_2 \equiv \beta_1, \beta \equiv \beta_2\} & \quad \text{C-APP} \\
(e_1 e_2)l \downarrow \beta, C_0 \cup C_1 \cup C_2, \Gamma_1 \cup \Gamma_2 \\
en \downarrow \tau, C, \Gamma & \quad \beta, \beta_x \text{ fresh} \\
\lambda x^{l_1} \rightarrow e \downarrow l_2 \downarrow \beta, C_0 \cup C, \Gamma \setminus x & \quad \text{C-ABS} \\
\text{e}_1 \downarrow \tau, C, \Gamma & \quad \text{e}_2 \downarrow \tau', C', \Gamma' & \beta, \beta_1, \beta_2 \text{ fresh} \\
\{\tau_1', \ldots, \tau_n'\} = \Gamma_2(x) & \quad k = \text{max}(1, n) \\
\tau_1, C_1, \Gamma_1 \ldots \tau_k, C_k, \Gamma_k \text{ fresh variants of } \tau, C, \Gamma \\
\Gamma'' = \Gamma_1 \cup \ldots \cup \Gamma_k \cup \Gamma' \setminus x & \quad C_0 = \{\beta \equiv l_2 \tau'\} \cup \{\tau_1' \equiv \tau_1, \ldots, \tau_n' \equiv \tau_n\} \\
C'' = C_0 \cup C_1 \cup \ldots \cup C_k & \quad \text{C-LET} \\
(\text{let } x^{l_1} = e_1 \text{ in } e_2)l_2 \downarrow \beta, C \cup C' \cup C'', \Gamma'' & \\
\text{let } x^{l_1} = e_1 \text{ in } e_2 \downarrow \beta, C \cup C' \cup C'', \Gamma'' & \\
(\Gamma_1 \cup \Gamma_2)(x) := \Gamma_1(x) \cup \Gamma_2(x) \\
\end{align*}
\]

Fig. 4. Constraint-based typing rules for Mini-ML.

same label, hence for the purpose of generating type error slices they could be combined into the single constraint \(\tau_1 \equiv \tau_2 \rightarrow \beta\).

The rules for C-LIT, C-BOT, C-VAR and C-ADD should now be straightforward. The result of C-LIT is a type variable, and not simply \text{Int}, so that it is possible to associate the label of the expression with a constraint. This is also done in rule C-VAR for the same reasons.

Abstraction has two associated labels hence rule C-ABS creates constraints annotated with different labels. Label \(l_1\) encodes the monomorphic nature of the variable binding. The constraints labelled with \(l_1\) accordingly ensure that each occurrence of the bound variable gets the same type. The label \(l_2\) encodes the requirement that the type of a lambda-abstraction is of function shape.
Rule C-Let looks a bit daunting, but the ideas are very similar to those of C-Abs. For each of the \( n \) use sites of the bound variable \( x \), the constraints generated from the expression \( e_1 \) are copied and connected to the corresponding type variables. This is equivalent to a capture avoiding substitution (i.e., inlining) of \( e_1 \) for every occurrence of \( x \) in \( e_2 \). It implements the requirement that every use site of \( x \) must be an instance of the (generalised) type of \( e_1 \) and is labelled with \( l_1 \). The constraint labelled with \( l_2 \) requires that the context must expect a type compatible with \( e_2 \)’s type.

The use of \( \max(1, n) \) ensures that there is at least one copy of the constraints generated for \( e_1 \). Thus, even if there is no use of \( x \) in \( e_2 \), \( e_1 \) has to type check on its own. Duplicating this many constraints will make a direct implementation of this constraint-generation scheme very inefficient. A more efficient implementation can be achieved by using let-bindings at the constraint level [7, 8].

4 Constraint-based Type Error Slicing

Type checking a program consists of translating it into a set of constraints and then solving those constraints. If the constraints are unsolvable it is possible to find a minimal unsolvable constraint set (MUCS). Because constraints are annotated with labels, we can trace each constraint to the source code location it originated from. A type error slice can now be constructed by taking all the labels in a MUCS and highlight the matching parts of the original program. For a formal description of how slices are constructed from a source program and a set of labels see [2].

The notion of minimality for determining a MUCS is as follows.

**Definition 1.** An unsolvable constraint set is minimal if removing any constraint from it makes it solvable.

```
1: function MINIMISE(C)
2:    L ← ∅
3:  for l ∈ labels(C) do
4:    C' ← Disable(l, C)  \( \triangleright \text{Disable}(l, C) = \{τ_1 \equiv \tau_2 \in C \mid l' \neq l\} \)
5:    if Solvable(C') then
6:      L ← L \cup \{l\}
7:    else
8:      C ← C'
9:  end if
10: end for
11: return L
12: end function
```

**Fig. 5.** Minimisation Algorithm. The input \( C \) is the set of constraints to minimise, the output is \( L \), the set of labels for a minimal constraint set.
As an example of this process consider the following very simple program:

\[(3^{l_1} \ 4^{l_2})^{l_3}\]

The superscripts denote the labels associated with each syntactic construct. From this program, the following unsolvable constraints are generated. Each constraint is annotated with a source code label.

\[\{\alpha \equiv_{l_1} \text{Int}, \beta \equiv_{l_2} \text{Int}, \alpha \equiv_{l_3} \beta \rightarrow \gamma\}\]

Removing the constraint \(\beta \equiv_{l_2} \text{Int}\) still leaves these constraints unsolvable, but if any other constraint is removed it becomes solvable. Therefore the only possible MUCS of this constraint set is \(\{\alpha \equiv_{l_1} \text{Int}, \alpha \equiv_{l_3} \beta \rightarrow \gamma\}\). After extracting the labels \((l_1, l_3)\) this gives the error slice.

\((3 \ldots)\)

For a given constraint set \(\mathcal{C}\) there may in general be multiple subsets \(\mathcal{D} \subseteq \mathcal{C}\) which satisfy this property. In practise it is not possible to always find all MUCS’s, as their number may be exponential in the program size [2]. However, it is feasible to find as many as MUCS possible within a fixed time period and then return, say, the smallest one [12].

In addition, only label sets are interesting for constructing type error slices. The MINIMISE function shown in Figure 5 therefore finds one minimal label set.

The idea of MINIMISE is to remove all constraints associated with a label and test whether the resulting constraint set is still solvable. If the constraint becomes solvable then (some of) the removed constraints must have been part of the MUCS. Otherwise, removing the constraints will not make the constraint set solvable, so they can be discarded.

### 4.1 Multiple Minimal Unsolvable Subsets

The algorithm in Figure 5 is non-deterministic because the order in which labels are picked from \(\text{labels}(\mathcal{C})\) is undefined. Furthermore, this algorithm will only find a single MUCS, but there may in fact be (and often are) multiple minimal unsolvable constraint sets. Consider the following constraint set:

\[\{\text{Int} \equiv_1 \alpha, \alpha \equiv_2 \beta, \beta \equiv_3 \gamma, \beta \equiv_4 \text{Char}, \gamma \equiv_5 \text{[Char]}\}\]

This constraint set has the three following minimal unsolvable subsets.

\[\{\text{Int} \equiv_1 \alpha, \alpha \equiv_2 \beta, \beta \equiv_4 \text{Char}\}\]
\[\{\text{Int} \equiv_1 \alpha, \alpha \equiv_2 \beta, \beta \equiv_5 \text{Char}\}\]
\[\{\beta \equiv_4 \text{Char}, \beta \equiv_5 \text{Char}\}\]

Which one is returned by the algorithm of Figure 5 depends on the order in which the constraints are explored. On the upside, the algorithm is fairly
efficient. Assuming Solvable is linear in the size of its input, then the algorithm is quadratic in the number of labels.

In order to find all minimal unsolvable subsets essentially all subsets of the input constraint set need to be explored. There is room for a few optimisations [12], but finding all MUCSs is not feasible in general as there may be exponentially many for a given constraint set [2].

Nevertheless, searching for several MUCSs (and therefore type error slices), perhaps coupled with a timeout, can be helpful to the user because slices may cover different program points. Program points that are part of many error slices may also indicate the most likely source of an error.

5 Source-based Type Error Slicing

As discussed in the introduction, constructing type error slices by first generating constraints and then minimising them is impractical. We now show how type error slices can be constructed without explicitly working on constraints.

On closer inspection it turns out that the algorithm of Figure 5 is actually independent of the constraint language used. If we can find suitable implementations of Solvable and Disable, then we can use MINIMISE to find a minimal label set.

If we have a type checker for expressions

\[ Typecheck : expr \rightarrow \{ \text{ok}, \text{error} \} \]

then we can use Typecheck in place of Solvable and use source expressions in place of constraints for the input parameter \( C \). Of course, both functions must be consistent, so the following axiom must hold.

**Axiom 1.** Let \( e \Downarrow \tau, C, \emptyset \) then \( Typecheck(e) = \text{ok} \) if and only if Solvable(\( C \)).

We also need a suitable version of Disable over expressions instead of constraints. Suitable in this context means that the constraints after source-level rewriting and the constraints obtained by directly removing constraints must be equivalent.

**Axiom 2.** If \( e \Downarrow \tau, C, \Gamma \) and Disable\( _E \)(\( l, e \)) \( \Downarrow \tau', C', \Gamma' \) and \( C'' = \text{Disable}(l, C) \), then Solvable(\( C'' \)) if and only if Solvable(\( C'' \)).

That is, the constraint-based type checking rules serve as a specification rather than an implementation. This includes in particular the mapping from locations to constraints which is essential to obtain accurate type error slices.

Figure 6 defines a rewriting function satisfying Axiom 2 for the Mini-ML language. Just like the rules of Figure 4 we have two rewrite rules for abstractions and let-bindings, one for each label (and a third if the target label is not part of the expression). For abstractions, label \( l_1 \) encodes to the monomorphic nature of the binding. The corresponding rewrite rule therefore shadows the binding.
We were interested in creating a type error slicer for Haskell. The following Section describes our experiences.

of \(x\) with a `let` and thus removes the constraint. Label \(l_2\) links the type of the context with the type of the body \(e\). By rewriting the body to \((\bot, e)\) this connection is broken thereby disabling the constraint. All constraints introduced by \(e\) are kept – merely constraints labelled with \(l_2\) are removed, as intended. The same technique is used for label \(l_1\) of a `let` expression which links the use sites of \(x\) with the type of \(e_1\).

**Theorem 1.** If we define \(\text{Disable}_E(l, e) = D[e]_l\) using the definition of \(D\) from Figure 6 then Axiom 2 holds.

**Proof.** By induction over the syntax of expressions and by strengthening the axiom to also allow open expressions (i.e., non-empty environments). This requires a number of properties about solvability of constraint sets, for example, that trivially satisfiable constraints can be dropped:

\[
\alpha \notin (fv(\tau) \cup fv(C)) \implies (\text{Solv}able(\{\alpha \equiv \tau\} \cup C) \iff \text{Solv}able(C))
\]

Using Axioms 1 \& 2 it is now easy to see that replacing `Solv`able by `Typecheck` and `Disable` by `Disable_E` as defined in `MINIMISE` gives an equivalent algorithm. We have successfully removed the dependency on a constraint-based type checker.

Whether it is possible to find a rewrite function satisfying Axiom 2 depends in part on the syntactic flexibility of the target programming language. We were interested in creating a type error slicer for Haskell. The following Section describes our experiences.
6 Type Error Slicing for Haskell

We have implemented a source-based type error slicer on top of the Glasgow Haskell Compiler (GHC). Haskell is quite a large language and contains several (main) syntactic categories: expressions, patterns, types, and statements (in do-notation). Implementing a full type checker for Haskell is a large undertaking and keeping up with GHC’s many extensions is even harder. Applying our technique to a full-fledged language like Haskell can therefore tell us more about whether the technique is indeed practical.

The main questions we were trying to answer were whether the additional syntactic categories would present problems, and how efficiently type error slices can be generated. We make the following observations:

- Performance is acceptable (i.e., seconds). Repeatedly calling the type checker with a full program is too expensive. We avoid this by using a simple pre-pass which consists of removing all definitions that are not involved in the error. Additionally, we treat all top-level type declarations as correct (if their definitions type check) and replace their definitions with `undefined`. As a result, the input to the type checker is generally small (only a single definition or recursive group).
- We can sometimes trade accuracy for implementation effort or performance. Instead of looking at every possible source location the slicer may remove bigger pieces of the program at once. (This is basically what SEMINAL does.)
- For each syntactic category (expressions, types, patterns, statements) we need a syntactic construct akin to `⊥`. Haskell, for example, does not have an equivalent of `⊥` at the level of types signatures. For GHC there is a workaround by using the “scoped type variables” extension.
  First, rewrite a definition with a type signature as shown in Figure 7. The ∀-quantifier binds all type variables rigidly in the body (i.e., it can only be unified with itself). Referring to a or b in the body refers to the same variable. If, however, a in the signature a -> b is replaced by c, then this c will be implicitly quantified. Together with the use of `unify`, c will now serve as a type-level bottom because it can be unified with anything.
  This is admittedly rather complex and we currently do not slice type signatures. Type signatures are either fully part of an error or not at all.
- We can only find error slices for certain kinds of errors, namely unification errors (“Cannot match ...”) and occurs check errors. Removing parts of the program may fix such an error, but may lead to an ambiguity error. For example, if part of the program contains the application `show x`, then rewriting this to `show ⊥` now causes an ambiguity error. We currently work around this issue by constructing slices for the same kind of error. That is, if disabling part of the program changes the error kind, we treat it as fixing the error and make it part of the error slice.

While type error slices can help narrow down the scope of an error, they do not usually give sufficient information about the cause of an error. For example,
Chameleon [10] uses constraints to infer types of sub-components of a minimal slice. In general, access to a constraint-based type checker can improve error messages of other kinds as well, such as ambiguity errors [12].

7 Future Work

The latest version of GHC now uses a constraint-based implementation, but it is still not suited to construct accurate type error slices.

Constraints may not properly be annotated with the original source locations, some constraints may be solved on-the-fly, and some constraints may correspond to several locations at once. Source-based type error slicing can help create accurate type error slices in this setting.

We plan to explore in what way we can exploit this knowledge about the type checker implementation for the purposes of type error slices. Closer integration with the type checker for example can improve performance by keeping track of which parts of the program have been processed by the type checker and thereby reduce the size of the input to the error slicer.

GHC’s type checker not only uses unification constraints, but also predicates (for type classes [11]), a restricted form of implication constraints [9], and more. We plan to explore how type error slices and the underlying idea of minimal unsolvable constraint set interacts with these new types of constraints.

8 Conclusion

We have shown how to construct type error slices by using a black-box type checker and a source-level rewriting informed by a constraint-based specification of the type system.

We have implemented a prototype for Haskell and were able to construct type error slices for some types of errors, namely unification errors and occurs check errors.

References


Abstract. Units of measure and type-level numbers are examples of type system extensions involving equational theories. Type inference for such an extension requires unification in a nontrivial theory. This complicates the generalisation step required for let-polymorphism in ML-style languages, as variable occurrence does not imply dependency. Previous work on units of measure (by Kennedy in particular) integrated free abelian group unification into the Damas-Milner type inference algorithm, but struggled with generalisation. I describe an approach to problem solving based on incremental minimal-commitment refinements in a structured context, and hence present abelian group unification and type unification algorithms which make type generalisation direct.

1 Introduction

Consider the following function to reverse a list and append another:

\[ \text{revApp} \ [\] \quad \text{as} = \text{as} \]
\[ \text{revApp} \ (x : xs) \text{as} = \text{revApp} \ xs \ (x : \text{as}) \]

What type should we give it? Instead of the usual Haskell type \([a] \rightarrow [a] \rightarrow [a]\), we could try to capture more information about the lengths of the lists involved, say \(\text{Vec} \ a \ m \rightarrow \text{Vec} \ a \ n \rightarrow \text{Vec} \ a \ (m + n)\). Now integer variables \(m\) and \(n\) appear in types, and must be compared in the appropriate equational theory: in order for the recursive call to be accepted, the typechecker must verify that \((m + 1) + n \equiv m + (n + 1)\). This may hold as a consequence of the definition of the \((+)\) function, or it may require some algebra. It is unlikely that the same definition of \((+)\) will make both \text{revApp} and concatenate \((+)\) typecheck directly.

Alternatively, consider this function, conventionally of type \(\text{Float} \rightarrow \text{Float}\):

\[ \text{distanceTravelled} \ \text{time} = \text{velocity} \ast \text{time} + (\text{acceleration} \ast \text{time} \ast \text{time}) / 2 \]
\[ \text{where} \ {\{ \text{velocity} = 2.0; \text{acceleration} = 3.6 \}} \]

Kennedy [6] teaches us how to enforce units of measure: with velocity in \(\text{ms}^{-1}\) and acceleration in \(\text{ms}^{-2}\), the system could infer the type \(\text{Float}(s) \rightarrow \text{Float}(m)\). The more specific types of both functions have subexpressions that must be compared in a more liberal equational theory than syntactic equality.

In previous work [4], McBride, McKinna and I described a rationalisation of syntactic unification and Hindley-Milner type inference in which term and type
variables live in a single dependency-ordered context. We solve problems in small, easily verified steps, each of which is most general. The additional structure in the context makes type generalisation for let-polymorphism particularly easy: we simply ‘skim off’ generalisable type variables from the end of the context.

Having applied our technique to the Hindley-Milner type system as a feasibility study, in this paper I extend the unification algorithm (and hence type inference) to support the equational theory of free abelian groups. This is a particularly fruitful choice as it retains decidable principal type inference. It is not the end of the story, but rather an example of the value of our approach to problem solving: that it gives a clearer account of the subtle issues surrounding generalisation.

Vytiniotis et al. [16] argue that “let should not be generalised” because of the difficulties generalisation presents in their setting (a complex equational theory including type-level functions and GADT local equality constraints). They may well be right; but perhaps a better account of generalisation will help us decide.

Kennedy [6–8] elegantly uses abelian groups to model units of measure with support for polymorphism, and has introduced them into the functional programming language F# [15]. However, he encounters problems with the approach to let-generalisation in the Damas-Milner type inference algorithm. It uses the occur-check to identify generalisable variables (those that are free in the type but not the environment), but variable occurrence does not imply variable dependency for the equational theory of abelian groups. Later we will see another way of looking at this: given the equation $\alpha \equiv \tau$, where $\alpha$ is a variable and $\tau$ is a type, the solution $\alpha := \tau$ is not necessarily most general!

He gives the example [8, p. 292, in slightly different notation]

\begin{align*}
\lambda x. \text{let } d := \text{div } x \text{ in } (d \text{ mass}, d \text{ time}), & \text{ where} \\
\text{div} :: \forall ab. F(ab) \rightarrow F(a) \rightarrow F(b), & \text{ mass ::= } F(\langle kg \rangle), & \text{ time ::= } F(\langle s \rangle).
\end{align*}

A na"ıve extension to the Damas-Milner algorithm fails to infer a type for this term, because polymorphism is lost: $d$ is given the monotype $F(a) \rightarrow F(ca^{-1})$ where $a$ and $c$ are unification variables, and $a$ cannot unify with $kg$ and $s$. However, if $d$ is given its principal type scheme $\forall a. F(a) \rightarrow F(ca^{-1})$, then the original term can be given type $F(c) \times F(c^{kg^{-1}} \times F(c^{s^{-1}})$ (see Section 5).

One possible solution to this problem is Kennedy’s notion of generaliser, “a substitution that ‘reveals’ the polymorphism available under a given type environment” [7, p. 23]. Such a substitution preserves types in the context (up to the equational theory) but permutes group variables so that the Damas-Milner generalisation rule can be used. Calculating a generaliser is specific to the equational theory, technically nontrivial, and not implemented in F#:

\begin{verbatim}
> fun x -> let d y = x / y in (d mass, d time) ;;
-----------------------------------------^^^^
error FS0001: Type mismatch.
Expecting a float<kg> but given a float<s>
The unit of measure 'kg' does not match the unit of measure 's'
\end{verbatim}
In the algorithm I will describe, insufficiently general unification shows up clearly as the source of this problem, and it has a correspondingly straightforward solution. With more structure in the context than just a set of typing assumptions, it is easier to see where generality can be lost, and we can prevent the loss of polymorphism in the first place, rather than trying to recover it after the fact. Maintaining generality explains the need for a new algorithm for abelian group unification: while the problem can be reduced to linear integer constraint solving, standard algorithms do not preserve the properties we care about!

Many authors have proposed designs for systems of units of measure, and relationships with linear algebra are well-established [5]. Following Kennedy, I will use integer powers, so units form a free abelian group. Some authors use rational powers (giving a vector space), including Rittri [13], who discusses the merits of both approaches. Chen et al. [3] give a useful overview of work on units of measure, and describe an alternative approach using static analysis.

Several impressive implementations of units of measure use advanced type system features such as GHC Haskell extensions [1] and C++ templates [14]. However, the difficulty of expressing a nontrivial equational theory at the type level means that they are complex, have limited inference capabilities and tend to expose the internal implementation in unfriendly error messages. Making units a type system extension, as in F#, results in a much more user-friendly system.

Rémy [12] extends the ML type system with other equational theories, using ranked unification to achieve easy generalisation; he does not address theories for which variable occurrence does not imply dependency, such as that of abelian groups. Variable ranking (as formalised by Kuan and MacQueen [9]) is also used in many ML type checkers for efficient generalisation; the algorithm described in the present paper implicitly manages ranks, by permuting the context.

In this paper, I describe a framework for contextual problem solving (Section 2) and an algorithm for abelian group unification in this framework (Section 3). Using this, I extend type unification to handle units of measure (Section 4), identifying a refinement needed for most general results. I sketch the corresponding type inference algorithm (Section 5) and conclude with some possible future directions (Section 6). A Haskell implementation of the algorithms is available at http://personal.cis.strath.ac.uk/~adam/units-of-measure/.

Acknowledgments I thank Conor McBride for helpful supervision and (along with James McKinna) for collaboration on previous work. This work was supported by Microsoft Research through its PhD Scholarship Programme.

2 Unification in context

Let me begin by defining the interconnected notions of context and contextualised statement. Informally, a well-formed context is one in which every declaration is explained by those which precede it. The declarations in a context
induce an equational theory, so we can consider how to evolve a context to solve an equation. This requires a notion of ‘information increase’ between contexts, capturing legitimate steps towards a solution. A solution is most general if all others can be obtained from it by information increases.

A context is a list of variable declarations; I write $\emptyset$ for the empty context and let $\Gamma, \Delta, \Theta$ range over contexts. Variables come in different sorts: $TY$ for syntactic type variables, $GR$ for group variables and $TM$ for term variables; all are bound in a single context. I write $\mathcal{V} = \{TY, GR, TM\}$ for the set of sorts. Let $\mathcal{V}_T$ be a distinct sets of variables for each sort $T \in \mathcal{V}$. I use $x$ as a variable of any sort or sort $TM$; $\alpha, \beta, \gamma$ for sort $TY$ (or one of $TY$ or $GR$) and $a, b, c$ for sort $GR$ only.

Statements are assertions that can be judged in contexts. Write $\Gamma \vdash S$ if statement $S$ holds in context $\Gamma$. The statement forms we will consider are:

$$S ::= \textrm{valid} \quad \text{the context is well-formed};$$

$$| S \land S' \quad \text{both statements } S \text{ and } S' \text{ hold};$$

$$| e \equiv_T e' \quad e \text{ and } e' \text{ are equivalent expressions of sort } T.$$  

I regard $\Gamma \vdash \cdot \equiv_T \cdot$ as a partial equivalence relation, so it is reflexive on well-formed expressions, and write $e$ is $T$ for $e \equiv_T e$. Thus $\tau$ is a well-formed type in $\Gamma$ if $\Gamma \vdash \tau$ is $TY$. A statement is well-formed if it contains well-formed expressions.

A declaration $xD$ in a context assigns a property $D$ to a variable $x$. For each sort $T$, we must describe the set of properties that can be assigned, and explain when they are valid context extensions by giving a validity map $ok_T$ from properties to statements. We must also explain what declarations mean by giving an interpretation map $[\cdot]_T$ from declarations of sort $T$ to statements.

For $T \in \{TY, GR\}$, variables may either be unknown or defined, and a declared variable is a well-formed expression that is equal to its definition, if any:

$$D ::= ? \mid ::= e, \quad ok_T (?) \mapsto \text{valid}, \quad [\alpha ?]_T \mapsto \alpha \text{ is } T,$$

$$ok_T ( ::= e ) \mapsto e \text{ is } T, \quad [\alpha ::= e]_T \mapsto \alpha \equiv_T e.$$  

Note that from $\alpha \equiv_T e$ we will be able to conclude $\alpha$ is $T$ (i.e. $\alpha \equiv_T \alpha$) by symmetry and transitivity. I will introduce the sort $TM$ in Section 5.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context validity</th>
<th>Lookup</th>
<th>Conjunction</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td></td>
</tr>
<tr>
<td>$\Gamma, xD \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td></td>
</tr>
<tr>
<td>$x \in \mathcal{V}_T \setminus \mathcal{V}_T (\Gamma)$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td></td>
</tr>
<tr>
<td>$\emptyset \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td>$\Gamma \vdash \text{valid}$</td>
<td></td>
</tr>
</tbody>
</table>

Note that from $\alpha \equiv_T e$ we will be able to conclude $\alpha$ is $T$ (i.e. $\alpha \equiv_T \alpha$) by symmetry and transitivity. I will introduce the sort $TM$ in Section 5.

Fig. 1. Rules for context validity, lookup, conjunction and equivalence
Figure 1 gives rules to construct a valid context and look up properties of variables in the context. Note that validity of properties $\text{ok}_T D$ is used to establish validity of the context, whereas the interpretation $\llbracket xD \rrbracket_T$ holds by LOOKUP if the declaration $xD$ is found in the context. I will discuss $;$ in Section 2.1.

The figure also gives rules to prove conjunctions and make $\Gamma \vdash \cdot \equiv_T \cdot$ an equivalence relation on well-formed expressions. The sort $TY$ of types has a binary constructor $\rightarrow$ for function types. I omit the context when it is constant.

I write $V_T(\Gamma)$ for the variables that are bound in the context $\Gamma$. The rules ensure that a valid context has no duplicated variables. This set is different from the set of free variables in a context suffix or expression $X$, which we write $\text{FV}_T(X)$. Thus $V_T(\alpha?, \beta := \alpha) = \{\alpha, \beta\}$ while $\text{FV}_T(\beta := \alpha) = \{\alpha\}$. A valid context defines all the variables it refers to, so it has no free variables! Or, to put it another way, free variables of an expression are those bound in its context.

For example, $\Gamma_0 = \alpha?, \beta := \alpha \rightarrow \alpha$ is a valid context and $\Gamma_0 \vdash \beta \equiv \alpha \rightarrow \alpha$ by LOOKUP. However, $\beta := \alpha, \alpha?$ is not a valid context because $\beta$ is not well-defined.

Derivations possess a monadic substitution structure analogous to that of expressions: the LOOKUP axiom is to derivations as variables are to expressions.

### 2.1 Solving problems by increasing information

Problem solving requires evolving the context in which a problem is posed into a context in which it is solved. For example, a unification problem consists of a context and two well-formed expressions; a solution must evolve the context to equate the expressions. So what are the legal evolutions of contexts?

A substitution $\delta$ from $\Gamma$ to $\Delta$ is given by maps $\delta_T : V_T(\Gamma) \rightarrow \{e \mid \Delta \vdash e \text{ is } T\}$ from variables in $\Gamma$ to well-formed expressions over $\Delta$ for each sort $T \in \mathcal{T}$. This substitution can be applied to a well-formed expression $e$ (or statement $S$) over $\Gamma$, replacing every variable $x$ of sort $T$ with $\delta_T(x)$ to give a well-formed expression $\delta e$ (or statement $\delta S$) over $\Delta$.

If $\delta$ is a substitution from $\Gamma$ to $\Delta$, and $\theta$ is a substitution from $\Delta$ to $\Theta$, then $\theta \cdot \delta$ is the substitution from $\Gamma$ to $\Theta$ given by $(\theta \cdot \delta)_T(x) = \theta(\delta_T(x))$ for $x \in V_T(\Gamma)$. Equivalence of substitutions is considered up to the equational theory, comparing values at all variables in the source context: if $\delta$ and $\theta$ are substitutions from $\Gamma$ to $\Delta$ then $\delta \equiv \theta$ means $\forall T \in \mathcal{T}. \forall x \in V_T(\Gamma). \Delta \vdash \delta x \equiv_T \theta x$.

Substitutions let us move from one context to another, but a legitimate evolution of a context must also preserve information in the context. In particular, the interpretation $\llbracket xD \rrbracket_T$ of a context entry $xD$ must hold in the new context.

However, we must also keep track of the order in the context, while allowing some permutation to deal with dependencies. I delimit localities within the context using $;$ separators. These will be placed by the type inference algorithm when inferring the type of a let-definition, so it can be generalised over the declarations in the locality. Making a context entry less local (moving it from the right to
the left of a separator) reduces the ability to generalise over it, so should be done only when essential for solving the problem. On the other hand, it is never possible to make a context entry more local (move it left to right).

Let \( \downarrow \) be the partial function from contexts and natural numbers to contexts such that \( \Gamma \downarrow n \) is \( \Gamma \) truncated after \( n \) occurrences of \( \#: \) separators, that is,
\[
\Xi_0 \#: \Xi_1 \#: \cdots \#: \Xi_m \downarrow n \mapsto \begin{cases} 
\Xi_0 \#: \cdots \#: \Xi_n, & \text{if } n \leq m, \\
\text{undefined}, & \text{if } n > m.
\end{cases}
\]

A substitution \( \delta \) from \( \Gamma \) to \( \Delta \) is an information increase, written \( \delta : \Gamma \sqsubseteq \Delta \), if for all \( n \in \mathbb{N} \) with \( xD \in \Gamma \downarrow n \), we have that \( \Delta \downarrow n \) is defined and \( \Delta \downarrow n \vdash \delta [xD]_\tau \).

I write \( \Gamma \sqsubseteq \Delta \) if \( \delta \) is the identity substitution \( \iota \).

The idea is that the localities of \( \Gamma \) and \( \Delta \) line up, and definitions in a locality of \( \Gamma \) hold as equations in the corresponding locality of \( \Delta \). An example increase is \( \alpha \# \beta? \sqsubseteq \beta? \), but on the other hand, \( \beta? , \alpha := \beta? \# \Xi \sqsubseteq \alpha \# \beta? \) as the first locality of the new context does not support \( \beta \) is \( \text{TY} \) or \( \alpha \equiv \beta \).

A context \( \Gamma \) and well-formed expressions \( d \) and \( e \) form a unification problem \( d \equiv \tau e \). A solution is a context \( \Delta \) and an information increase \( \delta : \Gamma \sqsubseteq \Delta \) such that \( \Delta \vdash \delta d \equiv \tau \delta e \). We say this solution is minimal or most general if every other solution \( \theta : \Gamma \sqsubseteq \Theta \) factors through it, i.e. there is a substitution \( \zeta : \Delta \sqsubseteq \Theta \) such that \( \theta \equiv \zeta \cdot \delta \). If the identity substitution is minimal, write \( \Gamma \hat{\sqsubseteq} \Delta \vdash d \equiv \tau e \).

For example, in the context \( \alpha \# \beta? \), the type unification problem \( \alpha \equiv \text{TY} \beta \to \beta \) has minimal solution \( \alpha \# \beta? \hat{\sqsubseteq} \beta?, \alpha := \beta \to \beta\# \vdash \alpha \equiv \text{TY} \beta \to \beta \).

We must ensure that statements we consider are stable: if \( S \) holds in a context, \( \delta S \) must hold after an information increase \( \delta \). This is easy to ensure by construction: we use only the \textsc{ Lookup} rule to extract information from the context. Once a problem is expressed as a stable statement, we can solve it using a minimal commitment strategy, making the smallest information increases possible until the problem is solved. Thanks to stability, this strategy delivers most general solutions. This is essentially McBride’s “optimistic optimisation” strategy [10].

3 Abelian group unification

Let us consider unification problems for abelian groups in the framework. A group expression \( d \) (with constants in \( K \)) is an expression of sort \( \text{GR} \) given by
\[
d ::= a \mid k \mid 0 \mid d + d \mid -d,
\]
where \( a \in V_{\text{GR}} \) and \( k \in K \). As shown in Figure 2, I extend the rules for equivalence of expressions given in Figure 1 by reflexivity and congruence (making group expressions well-formed), together with the four abelian group axioms of commutativity, associativity, inverses and identity. I write \( a, b, c \) for group variables; \( d, e, f \) for group expressions and \( m, n \) for integers.
Fig. 2. Declarative rules for group expression equivalence

Let $nd$ mean $d$ added to itself $n$ times, $-(nd)$ mean $d$ mean $d−e$ mean $d+(-e)$. Group expressions have a normal form $\sum_i n_i d_i$ where the $n_i$ are nonzero integers and the $d_i$ are distinct atoms (variables or constants) sorted in some order. For example, the expression $a + a + b + 0 + b + a$ has normal form $3a + 2b$.

Consider the equation $3a + 2b \equiv 0$ in the context $a?,b?$. Since $2 \nmid 3$, we cannot simply solve for $b$, but we can simplify the problem by setting $b := c − a$ where $c$ is a fresh variable. This gives $a + 2c \equiv 0$ in the context $a?,c?$, which can be solved by rearranging and taking $a := −2c \equiv 0$ to give $c?,a? := −2c,b := c − a$.

More generally, when solving such an equation, we will ask whether a variable has the largest coefficient, and if not, reduce the other coefficients by it to simplify the problem. Some notation is in order. Suppose $d \equiv \sum_i n_i d_i$ and define:

$$\maxc(d) = \max\{|n_i| \mid d_i \text{ is a variable}\}, \quad \text{highest absolute coefficient;}$$

$$Q_n(d) = \sum_i (n_i \text{ quot } n) d_i, \quad \text{quotient by } n \text{ of every coefficient;}$$

$$R_n(d) = \sum_i (n_i \text{ rem } n) d_i, \quad \text{remainder by } n \text{ of every coefficient;}$$

where $\cdot \text{ quot } \cdot$ is integer division truncated towards zero, and $\cdot \text{ rem } \cdot$ is the corresponding remainder, so for example $-3 \text{ quot } 2 = -1$ and $-3 \text{ rem } 2 = -1$. The important points, which I will make use of later, are that for every $d$,

$$nQ_n(d) + R_n(d) \equiv d \quad \text{and} \quad \maxc(R_n(d)) < n.$$

### 3.1 The abelian group unification algorithm

I must explain how to solve unification problems of the form $d \equiv_{\text{GR}} e$. Thanks to the inverse operation, it suffices to consider the equivalent matching problem $d − e \equiv 0$, which I will write $d − e \equiv \text{id}$.

Figure 3 shows the algorithm presented as a collection of inference rules. Given as input a context $\Gamma, \Psi$ and a group expression $d$, the judgment $\Gamma, \Psi \vdash d \equiv \text{id}$ means that the algorithm outputs the context $\Delta$ such that $\Delta \vdash d \equiv \text{id}$. Note that the rules are entirely syntax-directed: at most one rule applies for any possible initial context and group expression. They lead directly to an implementation.
So how does the algorithm work? If the problem is 0 id, then it is Trivial. Otherwise, we move back through the context, skipping over variables that do not occur in the problem (including type and term variables) using Ignore, and moving through localities using Repossess. When we encounter a defined variable, we must substitute it out (with Expand) to simplify the problem. The boxed suffix \( \Psi \) will either be empty (written \( \mathcal{E} \)) or contain only the unknown variable with the strictly largest coefficient in \( d \), if any. The Reduce and Collect rules move this variable back in the context, since there is no simplification that can usefully be applied to it. Other rules will insert the variable into the context when it no longer has the largest coefficient.

The interesting cases arise when we reach an unknown variable \( a \) that occurs in the problem, which we write as \( na + e \) id (always meaning that \( a \not\in \text{FV}_{\text{GR}}(e) \)). Suppose the normal form of \( e \) is \( \sum_i n_i e_i \). There are four possibilities, either:

1. \( n \mid n_i \) for all \( i \);
2. \( |n| \leq |n_i| \) for some \( i \) with \( e_i \) a variable, and the previous case does not apply;
3. \( |n| > |n_i| \) for all \( i \) with \( e_i \) a variable, but \( e \) has at least one variable; or
4. \( e \) has no variables.

**Case 1.** If \( n \mid n_i \) for all \( i \), then there is some \( f \) such that \( e \equiv nf \). The rule Define applies and we set \( a := -f \) to give \( na + e \equiv na + nf \equiv -nf + nf \equiv 0 \). This is clearly a solution, and it is most general for the free abelian group.
Case 2. If $|n| \leq |n_i|$ for some $i$ with $e_i$ a variable (but $n$ does not divide all the coefficients), then the Reduce rule applies and simplifies the problem by reducing the coefficients modulo $n$. Recall that $e \equiv nQ_n(e) + R_n(e)$ where $Q_n(e)$ is given by taking the quotient by $n$ of the coefficients in $e$. We can generate a fresh variable $b$ and define $a := b - Q_n(e)$, giving

$$na + e \equiv n(b - Q_n(e)) + e \equiv nb + (e - nQ_n(e)) \equiv nb + R_n(e).$$

Case 3. Here $|n| > |n_i|$ for all $i$, so neither of the two previous cases apply, but there are variables in $e$. Now $n$ is the largest coefficient of a variable, so reducing the coefficients modulo $n$ would leave them unchanged. Instead, we have to Collect $a$ and move it further back in the context. This rule maintains the invariant that $\Psi$ contains only the variable with the largest coefficient, if any; the invariant also guarantees that $\Psi$ will be empty when the rule applies.

Case 4. Finally, if there are no variables in $e$ then the problem is of the form $na + k \text{ id}$, where $k$ is a constant expression and $n \nmid k$, so it has no solution.

3.2 Correctness of the algorithm

I prove correctness for the matching problem $d \text{ id}$; correctness for the unification problem $d \equiv_{\text{GR}} e$ is a corollary. For details of the proofs, consult the appendix.

Lemma 1 (Soundness and generality of abelian group unification).
If unification succeeds with $\Gamma, \Psi \rightarrow \Delta \vdash d \text{ id}$, then $V_{\text{TV}}(\Gamma, \Psi) = V_{\text{TV}}(\Delta)$, $V_{\text{GR}}(\Gamma, \Psi) \subseteq V_{\text{GR}}(\Delta)$ and it gives a most general solution $\Gamma, \Psi \overset{\wedge}{\rightarrow} \Delta \vdash d \equiv_{\text{GR}} 0$.

Proof (Sketch). By structural induction on derivations. Each step preserves the meaning of the problem, so the result is a solution (soundness). Moreover, each step makes commitments only if they are essential to solving the problem, so the result is most general. The interesting part is proving generality of the Repossess rule, since this involves moving $\Psi$ into a new locality, which could restrict the solution. However, if $\Psi$ contains a variable then it has the strictly largest coefficient, so the problem can be solved only by moving this variable.

Lemma 2 (Completeness of abelian group unification).
If $d$ is a well-formed group expression in $\Gamma$, and there is some $\theta : \Gamma \sqsubseteq \Theta$ such that $\Theta \vdash \theta d \equiv_{\text{GR}} 0$, then the algorithm produces $\Delta$ such that $\Gamma, [\Theta] \rightarrow \Delta \vdash d \text{ id}$.

Proof (Sketch). We can show that the algorithm terminates by exhibiting a termination metric, so we are justified in reasoning by structural induction on the call graph. Completeness is by the fact that the rules cover all solvable cases and preserve solutions, so if no rule applies then the original problem can have had no solutions. This occurs if a non-unit constant is equated to 0 or there is only one variable and its coefficient does not divide the coefficient of one of the constants (e.g. $2a + k \text{ id}$).
Decompose
\[ \Gamma \rightarrow \Delta \vdash \tau \equiv \upsilon \]

Idle
\[ T, \alpha D \rightarrow T, \alpha D \vdash \alpha \equiv \alpha \]

Define
\[ \Gamma, \alpha ? \rightarrow \Gamma, \alpha := \beta \vdash \alpha \equiv \beta \]

Expand
\[ \Gamma \rightarrow \Delta \vdash \alpha \equiv \beta \]
\[ T, \alpha := \tau \rightarrow \Delta, \alpha := \tau \vdash \alpha \equiv \beta \]

Ignore
\[ \Gamma \rightarrow \Delta \vdash \alpha \equiv \beta \]
\[ T, x D \rightarrow \Delta, x D \vdash \alpha \equiv \beta \]

Skip
\[ \Gamma \rightarrow \Delta \vdash \alpha \equiv \beta \]
\[ T \upharpoonright \rightarrow \Delta \upharpoonright \vdash \alpha \equiv \beta \]

Solve
\[ \Gamma \mid \Xi \rightarrow \Delta \vdash \alpha \equiv \tau \]
\[ \tau \text{ not variable} \]

Define
\[ \alpha \notin \text{FV}_{TY}(\tau, \Xi) \]
\[ \Gamma, \alpha ? \mid \Xi \rightarrow \Gamma, \Xi, \alpha := \tau \vdash \alpha \equiv \tau \]

Expand
\[ \Gamma \mid \Xi \rightarrow \Delta \vdash \upsilon \equiv \tau \]
\[ \alpha \notin \text{FV}_{TY}(\tau, \Xi) \]
\[ \Gamma, \alpha := \upsilon \mid \Xi \rightarrow \Delta, \alpha := \upsilon \vdash \alpha \equiv \tau \]

Ignore
\[ \Gamma \mid \Xi \rightarrow \Delta \vdash \alpha \equiv \tau \]
\[ \alpha \notin \text{FV}_{TY}(\tau, \Xi) \]
\[ \Gamma \mid \Xi \rightarrow \Delta \vdash \alpha \equiv \tau \]
\[ \alpha \notin \text{FV}_{TY}(\tau, \Xi) \]

Depend
\[ \Gamma \mid x D, \Xi \rightarrow \Delta \vdash \alpha \equiv \tau \]
\[ \alpha \notin \text{FV}_{TY}(\tau, \Xi) \]
\[ \Gamma \mid x D \mid \Xi \rightarrow \Delta \vdash \alpha \equiv \tau \]
\[ x \notin \text{FV}_{TY}(\tau, \Xi) \]

Repossess
\[ \Gamma \mid \Xi \rightarrow \Delta \vdash \alpha \equiv \tau \]
\[ \Gamma \mid \Xi \rightarrow \Delta \vdash \alpha \equiv \tau \]

Symmetrical variants of Define, Expand and Solve omitted.

Fig. 4. Original algorithmic rules for type unification

4 Unification for types with units of measure

Having developed a unification algorithm for the theory of abelian groups, let us extend type unification to support units of measure. The unification algorithm from my previous work [4] is shown in Figure 4. As in the group unification algorithm in Section 3, the rules are entirely syntax directed and lead immediately to an implementation. There are two kinds of rule:

- ‘Unify’ steps start the process: given an input context \( \Gamma \) and well-formed types \( \tau \) and \( \upsilon \), the judgment \( \Gamma \rightarrow \Delta \vdash \tau \equiv \upsilon \) means that the unification problem \( \tau \equiv_{TY} \upsilon \) is solved with output context \( \Delta \).
- ‘Solve’ steps handle flex-rigid unification problems\(^1\): given a context \( \Gamma, \Xi \), a type variable \( \alpha \) in \( \Gamma \) and a well-formed non-variable type \( \tau \) in \( \Gamma, \Xi \), the judgment \( \Gamma \mid \Xi \rightarrow \Delta \vdash \alpha \equiv \tau \) means that the problem \( \alpha \equiv_{TY} \tau \) is solved with output context \( \Delta \). The context suffix \( \Xi \) collects type or group variable declarations that \( \tau \) depends on but that cannot be used to solve the problem.

\(^1\) Recall that a flex-rigid problem is to unify a variable and a non-variable expression; a flex-flex problem has two variables and a rigid-rigid problem has two non-variables.
The algorithm starts by applying the Decompose rule to split up rigid-rigid problems into subproblems and solve them sequentially. If a flex-flex problem \( \alpha \equiv \beta \) is reached, the input context is searched for \( \alpha \) and \( \beta \), moving past other entries with Ignore or Skip. When either variable is found, the problem is either ignored by Idle as trivial, solved by Define if the variable is unknown, or simplified by Expand substituting out the definition.

If a flex-rigid problem \( \alpha \equiv \tau \) is reached, the Solve rule applies. Now the context is searched as in the flex-flex case, except that a list \( \Xi \) of hereditary dependencies of \( \tau \) (either type or group variables) must be accumulated. These must be moved back in the context until Defines (solve \( \alpha \) with \( \tau \)) or Expands (substitute out \( \alpha \)) applies. Note the occur check performed by both these rules: if \( \alpha \in FV_{\text{ty}}(\tau, \Xi) \) then \( \alpha \equiv \tau \) has no solutions. The suffix \( \Xi \) may be moved into a previous locality by Repossess, making its entries less generalisable, so Depends only adds entries to it if necessary; otherwise ignores skips them.

For example, consider the context \( \beta?, \alpha? ;; \gamma? \) and constraint \( \alpha \equiv \beta \rightarrow \gamma \). Since this is a flex-rigid problem the Solve rule applies, followed by Depends as \( \gamma \) appears in the type. The Repossess rule moves into the previous locality, making the accumulated \( \gamma \) less generalisable. Finally, Defines applies to solve the constraint giving the final context \( \beta?, \gamma?, \alpha := \beta \rightarrow \gamma? \).

### 4.1 Units of measure as an abelian group

A unit (of measure) is a group expression with constants in some set of base units. For familiarity’s sake, I write units in multiplicative notation, with identity 1: if \( a, b \) are variables and \( m, s \) are base units, then \( abm^2s^{-1} \) is a derived unit.

Let us extend the language of types with a single new type \( F\langle d \rangle \) of numbers parameterised by units, adding a congruence rule to the declarative system and a corresponding algorithmic rule that invokes abelian group unification:

\[
\begin{align*}
d \equiv_{GR} e \\
F\langle d \rangle \equiv_{TV} F\langle e \rangle
\end{align*}
\]

**Unit**

\[
\begin{align*}
I, [E] \vdash \Delta \vdash de^{-1} \text{id} \\
I \vdash \Delta \vdash F\langle d \rangle \equiv F\langle e \rangle
\end{align*}
\]

Suppose the algorithm is used to solve \( F\langle bc \rangle \rightarrow \alpha \equiv F\langle b \rangle \rightarrow F\langle c \rangle \) in the context \( b?, \alpha?, c? \). First the constraint \( F\langle bc \rangle \equiv F\langle b \rangle \) is reduced to \( bc \equiv_{GR} b \) by Unit, and this is solved by group unification (Section 3) to give \( b?, \alpha?, c := 1 \). Then the constraint \( \alpha \equiv F\langle c \rangle \) is solved by moving \( c \) to give \( b?, c := 1, \alpha := F\langle c \rangle \).

Thus we have a unification algorithm for types, but is it correct? It certainly ought to be sound and complete, because the new algorithmic rule corresponds directly to the declarative rule. However, we shall see that generality fails.
4.2 Loss of generality and how to recover it

Suppose we seek $\alpha \equiv F(b_0 b_1)$ in the context $\alpha ? b_0 ?, b_1 ?$. Following the algorithm, this flex-rigid problem is solved by moving $b_0$ and $b_1$ into the previous locality, and instantiating $\alpha$, resulting in the context $b_0 ?, b_1 ?, \alpha := F(b_0 b_1)$. However, a more general solution exists, namely $c ?, \alpha := F(c) \# b_0 ?, b_1 := cb_0^{-1}$, where $c$ is a fresh group variable and $b_0$ is still local. Why did the algorithm fail to find this?

The syntactic equational theory has the property that equivalent expressions have the same sets of free variables. Indeed, some other useful theories share this property [12]. However, it does not hold for the theory of abelian groups: for example, the equation $a a^{-1} \equiv 1$ has $a$ free on the left only. Thus variable occurrence does not imply dependency. The syntactic occurs check performed by the unification algorithm is too hasty.

The problem is that, when solving a flex-rigid constraint, we do not actually know that the variable must be syntactically equal to the type: units need be equal only in the equational theory of abelian groups. We can decompose such constraints into a flex-rigid constraint on type variables, with fresh variables in place of units, and additional constraints to make the fresh variables equal to the units. A rigid type decomposes into a ‘hull’, or ‘type skeleton’\(^3\), that must match exactly, and a collection of constraints in the richer equational theory.

In our example, the constraint $\alpha \equiv F(b_0 b_1)$ becomes $\alpha \equiv_{TV} F(c) \wedge c \equiv_{GR} b_0 b_1$ in context $\alpha ? b_0 ?, b_1 ?, c ?$. After solving the first part we have $c ?, \alpha := F(c) \# b_0 ?, b_1 ?$, and solving the second yields the principal solution $c ?, \alpha := F(c) \# b_0 ?, b_1 := cb_0^{-1}$.

I write $\rho \langle - \rangle$ for the hull of the type $\rho$, parameterised by a vector of units: $\rho = F(d) \rightarrow F(e)$ has hull $\rho \langle - \rangle = F\langle - \rangle \rightarrow F\langle - \rangle$ and $\rho \langle \vec{a} \rangle = F\langle a_0 \rangle \rightarrow F\langle a_1 \rangle$.

Let us modify the rules to maintain the invariant that the only group variables a flex-rigid problem depends on (i.e. those in the rigid type $\tau$ or suffix $\Xi$) are fresh unknowns. This ensures group variables are never made less local by collecting them in $\Xi$ as dependencies. Type unification does not prejudice locality of group variables: that is up to the group unification algorithm! The SOLVE and DEPENDS rules are replaced by the following modified versions:

\[
\text{SOLVE'}
\begin{align*}
\Gamma | \vec{b} & \rightarrow \Delta_0 \vdash \alpha \equiv \rho\langle \vec{b} \rangle \\
\Gamma & \rightarrow \Delta \vdash \alpha \equiv \rho\langle \vec{e} \rangle \\
\hline
\Gamma & \rightarrow \Delta \vdash \vec{b} \equiv_{GR} \vec{e} \\
\end{align*}
\rho \text{ not variable, } \vec{b} \text{ fresh}
\]

\[
\text{DEPENDS'}
\begin{align*}
\Gamma | \beta ?, \Xi & \rightarrow \Delta_0 \vdash \alpha \equiv \tau \\
\Gamma, \beta ? | \Xi & \rightarrow \Delta \vdash \alpha \equiv \tau \\
\hline
\alpha \neq \beta, \beta \in \text{FV}_T(V, \Xi)
\end{align*}
\]

\(^2\) Such equational theories sometimes described as regular [2], but we avoid this term because it means too many different things in other contexts.

\(^3\) This term was suggested by an anonymous reviewer of a previous version.
Depends$''$

\[
\Gamma | \vec{b}, \beta := \rho(\vec{b}), \Xi \rightarrow \Delta_0 \vdash \alpha \equiv \tau \quad \Delta_0 \rightarrow \Delta \vdash \vec{b} \equiv_{GR} \vec{e} \quad \alpha \neq \beta, \vec{b} \text{ fresh,} \\
\beta \in \text{FV}_{TY}(\tau, \Xi)
\]

We solve vectors of equations one at a time, threading the context:

\[
\Delta_0, [\vec{e}] \rightarrow \Delta_1 \vdash b_1 e_1^{-1} \text{id} \quad \ldots \quad \Delta_{n-1}, [\vec{e}] \rightarrow \Delta_n \vdash b_n e_n^{-1} \text{id}
\]

\[
\Delta_0 \rightarrow \Delta_n \vdash b_1, \ldots, b_n \equiv_{GR} e_1, \ldots, e_n
\]

4.3 Correctness of type unification

With the above refinement, type unification gives most general results.

Lemma 3 (Soundness and generality of type unification).

(a) If type unification succeeds with \(\Gamma \rightarrow \Delta \vdash \tau \equiv \upsilon\), then \(V_{TY}(\Gamma) = V_{TY}(\Delta)\), \(V_{GR}(\Gamma) \subseteq V_{GR}(\Delta)\) and it gives a most general solution \(\Gamma \hat{\leq} \Delta \vdash \tau \equiv \upsilon\).

(b) Correspondingly, if \(\Gamma | \Xi \rightarrow \Delta \vdash \alpha \equiv \tau\), then \(V_{TY}(\Gamma, \Xi) = V_{TY}(\Delta)\), \(V_{GR}(\Gamma) \subseteq V_{GR}(\Delta)\) and \(\Gamma, \Xi \hat{\leq} \Delta \vdash \alpha \equiv \tau\).

Proof (Sketch). By structural induction on derivations, as in Lemma 1, noting that each step preserves solutions and follows a minimal commitment strategy. The new rules in Section 4.2 ensure the type \(\tau\) in the flex-rigid problem \(\alpha \equiv \tau\) contains only group variables, not compound units. When a ; separator is found, any solution must move all the dependencies into the previous locality. \(\Box\)

Lemma 4 (Completeness of type unification).

(a) If the types \(\upsilon\) and \(\tau\) are well-formed in \(\Gamma\) and there is some \(\Theta : \Gamma \subseteq \Theta\) such that \(\Theta \vdash \theta \upsilon \equiv \theta \tau\), then unification produces \(\Delta\) such that \(\Gamma \rightarrow \Delta \vdash \upsilon \equiv \tau\).

(b) Moreover, if \(\Theta : \Gamma, \Xi \subseteq \Theta\) is such that \(\Theta \vdash \theta \alpha \equiv \theta \tau\) and the following conditions are satisfied:

\(\alpha \in V_{TY}(\Gamma)\), \(\tau\) is not a variable,
\(\Gamma, \Xi \vdash \tau\) is TY, \(\Xi\) contains only type or group variable declarations
\(\beta \in V_T(\Xi) \Rightarrow \beta \in \text{FV}_T(\tau, \Xi)\);

then there is some context \(\Delta\) such that \(\Gamma | \Xi \rightarrow \Delta \vdash \alpha \equiv \tau\).

Proof (Sketch). As before, we show termination, then reason by structural induction. The rules preserve solutions, so if a recursive call fails then the whole problem must have no solution. The only cases not covered are rigid-rigid mismatches (such as unifying \(\upsilon \rightarrow \tau\) with \(F(\vec{d})\)) and occur-check failures (such as unifying \(\alpha\) with \(\alpha \rightarrow \alpha\)), neither of which have any solutions. \(\Box\)
5 Type inference

We have seen a unification algorithm for types containing units of measure, and this extends to a type inference algorithm for the corresponding type system. I will only sketch the extension here; it is detailed in the previous paper [4]. Besides the new unification algorithm and the ability to quantify over group variables, no changes to type inference are required.

A type scheme \( \sigma \) is a type quantified over by some context entries of sort TY or GR. For example, \( \forall \alpha \forall (\beta := \alpha).\alpha \to \beta \) corresponds to the type \( \alpha \to \beta \) quantified over by \( \alpha ? \), \( \beta := \alpha \). Unknown variables are universally quantified, whereas defined variables represent abbreviations that are stored in the type scheme. (For a more conventional presentation, they could be substituted out.)

Just as a type scheme quantifies over a context extension, so a statement can be conditional on an extension: if \( S \) is a statement, then so is \( \Xi S \), with \( \Gamma \vdash (\Xi S) \) iff \( \Gamma, \Xi \vdash S \) (omitting some freshness-related details). Let us introduce a new statement form \( t : \tau \) for type assignment, where \( t \) is a term and \( \tau \) is a well-formed type. We then define the scheme assignment statement \( t :: \forall \Xi.\tau \rightarrow (t : \tau) \).

Now let us add declarations of the form \( x :: \sigma \) to the context, where \( x \in \mathcal{V}_{TM} \) is a term variable and \( \sigma \) is a type scheme. Recall from Section 2 that we must define the validity map, to say when a property makes sense, and the interpretation map, to explain declarations as statements:

\[
ok_{TM}(:: \forall \Xi.\tau) \rightarrow \Xi \succ (\tau \text{ is TY}), \quad [x :: \sigma]_{TM} \rightarrow x :: \sigma.
\]

Thanks to the latter definition, the LOOKUP rule from Section 2 can be used to assign types to variables. The other rules for type assignment statements are given in Figure 5. These rules can be converted into an algorithm that is structurally recursive on terms, building up a context along the way:

- For a term variable \( x \), look up its type scheme in the context and expand the scheme with fresh variables to produce a type.
- For a lambda abstraction \( \lambda x. t \), create a fresh unknown type variable \( \beta \), add it with \( x :: \beta \) to the context, then infer the type of \( t \).
- For an application \( fa \), infer the types of \( f \) and \( a \), then appeal to unification to ensure \( f \) is a function whose domain corresponds to the type of \( a \).
- For a let binding \( \text{let } x := s \text{ in } w \) a few steps are required:
  1. place a marker \( # \) in the context, starting a new locality;
2. infer the type $\tau$ of $s$;
3. generalise $\tau$ over all type variables in the locality, producing a scheme $\sigma$;
4. extend the context with the new term variable $x$ having scheme $\sigma$; and
5. infer the type of $w$.

Generalisation is easy and there is no need to complicate the type inference algorithm to deal with units of measure. We can extend the initial context with constant terms that use the new types. Moreover, thanks to the refinement of Section 4.2, the algorithm copes naturally with the problematic term from Section 1, correctly inferring its most general type. Recall the example:

$$\lambda x. \text{let } d := \text{div } x \text{ in } (d \text{ mass}, d \text{ time}),$$

where

| $\text{div}$ | $\forall ab . F(ab) \rightarrow F(a) \rightarrow F(b)$, |
| $\text{mass}$ | $F(kg)$, |
| $\text{time}$ | $F(s)$.

With the new type inference algorithm, at the crucial point where the type of $d$ is to be generalised, we have the context, type and constraint

$$\gamma ?, x : \gamma ; a ?, b ? \vdash \text{div } x : F(a) \rightarrow F(b) \quad \text{subject to } \gamma \equiv F(ab).$$

(Here $\gamma$ is an unknown fresh type variable standing in for the type of $x$.) If $c$ is a fresh group variable, the constraint decomposes into two simpler constraints $\gamma \equiv_{\text{TV}} F(c) \land c \equiv_{\text{GR}} ab$, which can be solved one at a time to give the context $c ?, \gamma := F(c), x : \gamma ; a ?, b := cb^{-1}$. Generalising by ‘skimming off’ type variables in the locality (and substituting out the definition of $b$) gives the principal scheme

$$c ?, \gamma := F(c), x : \gamma \vdash d : \forall a. F(a) \rightarrow F(ca^{-1}).$$

### 6 Discussion

I have shown how to combine abelian group unification with syntactic unification in such a way that generalisation is straightforward. Unlike the usual Damas-Milner approach to generalisation, the structured context discipline adopted here works well with the nontrivial equational theory. The algorithms presented here solve unification problems by making gradual refinements towards a solution, so it is comparatively easy to check each step is sound and most general.

A key point is that flex-rigid equations $\alpha \equiv \tau$ cannot always be solved by instantiating $\alpha$ to $\tau$, in the presence of a nontrivial equational theory. Instead, $\tau$ decomposes into a ‘rigid hull’ (the outer structure that $\alpha$ must match exactly) and a collection of subexpressions (that must match in the equational theory).

I plan to apply this technique to other equational theories and more advanced type systems. In particular, I am interested in the computational equality of dependent types [10], and Miller’s ‘mixed prefix’ unification [11], which can be used to implement arbitrary-rank polymorphism as available in modern Haskell.
In the beginning I mentioned types indexed by integers, which form an abelian group under addition, so type inference could be implemented using the algorithm described here. However, for many purposes natural numbers are needed, so I am exploring how to solve inequalities in this setting. There are also many other algebraic structures to consider, notably rings and semirings, though unification is not often unitary for their equational theories.

In this paper we have been following the trail that Kennedy blazed, both in the representation of units of measure using a free abelian group with constants, and the observation that unification has decidable most general unifiers in this case. In order to extend the technique to less convenient type systems, we will need to deal with problems that cannot necessarily be solved on the first try. Where will we store problems that we cannot yet solve? In the context, of course!

References

Appendix

First I give a technical lemma that illustrates explicitly the reasoning used to show generality. Subsequent proofs will not be in quite this much detail.

**Lemma 5.** Suppose $d$ is a well-formed group expression in $\Gamma$, $\gamma : \Gamma \subseteq \Gamma_0$ is invertible (i.e. there exists $\gamma^{-1} : \Gamma_0 \subseteq \Gamma$ such that $\gamma \cdot \gamma^{-1} \equiv \iota$ and $\gamma^{-1} \cdot \gamma \equiv \iota$), $\delta : \Delta \subseteq \Delta_0$ is invertible and $\iota \cdot \gamma \equiv \delta \cdot \iota$. Then the following rule is admissible:

$$
\frac{\Gamma_0 \subseteq \Delta_0 \vdash \gamma d \equiv 0}{\Gamma \subseteq \Delta \vdash d \equiv 0}.
$$

**Proof.** We assume $\Gamma_0 \subseteq \Delta_0 \vdash \gamma d \equiv 0$. Now $\delta^{-1} \cdot \iota \cdot \gamma \equiv \iota : \Gamma \subseteq \Delta$, and $\Delta_0 \vdash \gamma d \equiv 0$ so $\Delta \vdash (\delta^{-1} \cdot \iota)(\gamma d) \equiv 0$ and hence $\Delta \vdash d \equiv 0$. For generality, let $\theta : \Gamma \subseteq \Theta$ be such that $\Theta \vdash \theta d \equiv 0$. Then $\theta \cdot \gamma^{-1} : \Gamma_0 \subseteq \Theta$ and $\Theta \vdash (\theta \cdot \gamma^{-1})(\gamma d) \equiv 0$ so by the assumption of minimality, there is a substitution $\zeta : \Delta_0 \subseteq \Theta$ such that $\theta \cdot \gamma^{-1} \equiv \zeta \cdot \iota$. Now $\zeta \cdot \delta \cdot \iota \equiv \zeta \cdot \iota \cdot \gamma \equiv \theta \cdot \gamma^{-1} \cdot \gamma \equiv \theta$, so $\zeta \cdot \delta : \Delta \subseteq \Theta$ is the required substitution. \qed

**Lemma 1 (Soundness and generality of abelian group unification).**

If unification succeeds with $\Gamma, [\Psi] \rightarrow \Delta \vdash d \ \text{id}$, then $\mathcal{V}_{\text{ty}}(\Gamma, \Psi) = \mathcal{V}_{\text{ty}}(\Delta)$, $\mathcal{V}_{\text{gr}}(\Gamma, \Psi) \subseteq \mathcal{V}_{\text{gr}}(\Delta)$ and it gives a most general solution $\Gamma, \Psi \models_{\text{gr}} \Delta \vdash d \equiv_{\text{gr}} 0$.

**Proof.** We proceed by structural induction on derivations. For soundness, it is easy to verify that $\Gamma, \Psi \subseteq \Delta$ and $\Delta \vdash d \equiv_{\text{gr}} 0$. Let us consider generality for each rule in Figure 3:

**Trivial** and **Ignore** are straightforward to check.

**Reduce, Collect and Expand** satisfy the generality condition by Lemma 5.

For **Define**, suppose $\theta : \Gamma, a? \Psi \subseteq \Theta$ is such that $\Theta \vdash \theta(na + ne) \equiv 0$. Then $\Theta \vdash n(\theta(a + e)) \equiv 0$ and hence $\Theta \vdash \theta(a + e) \equiv 0$, since we are working in the **free** abelian group. Thus $\Theta \vdash \theta a \equiv \theta(-e)$ and so $\theta : \Gamma, \Psi, a := -e \subseteq \Theta$.

Finally, we consider **Repossess**. If $\Psi$ is empty then the result is straightforward. Otherwise, it contains a single unknown variable $\beta$; let $d \equiv n\beta + e$. Suppose $\theta : \Gamma, \beta? \subseteq \Theta$; $\Phi$ is such that $\Theta \vdash \theta(n\beta + e) \ \text{id}$. Then $\Theta \vdash \theta(n\beta \equiv -(\theta e)$ but $\theta e$ is defined over $\Theta$ so $\theta \beta$ must be defined over $\Theta$ (by substituting out definitions in $\Phi$ if necessary). Thus $\theta : \Gamma, \beta? \subseteq \Theta$ and the result follows by the inductive hypothesis. \qed
Lemma 2 (Completeness of abelian group unification).
If \( d \) is a well-formed group expression in \( \Gamma \), and there is some \( \theta : \Gamma \sqsubseteq \Theta \) such that \( \Theta \vdash \theta d \equiv_{GR} 0 \), then the algorithm produces \( \Delta \) such that \( \Gamma, [\mathcal{E}] \rightarrow \Delta \vdash d \ id \).

Proof. First, let us establish termination of the rules when viewed as an algorithm, where hypotheses correspond to recursive calls. Termination is by the lexicographic order on the total length of the context (including \( \Psi \)), the maximum coefficient of a variable in the expression being unified, and the length of the first part of the context (excluding \( \Psi \)). Only the REDUCE and COLLECT rules do not decrease the total length on recursive calls; moreover, REDUCE decreases the maximum coefficient of a variable (to \( n \)) and COLLECT decreases the length of the first part of the context. Note that the final result may be longer than the original context, due to REDUCE.

Since the algorithm terminates, we are entitled to reason about completeness by induction on the call graph. By inspection of the rules, we observe that only two possible cases are not covered: either \( d \) is a non-zero constant expression, or \( d \) contains exactly one variable \( a \), and the coefficient of \( a \) does not divide the coefficients of the constants. In either case, there are no possible solutions of the unification problem \( d \equiv_{GR} 0 \).

Finally, we note that each rule preserves solutions: that is, if the initial problem (conclusion of the rule) has a solution then the rewritten problem (hypothesis of the rule) must also have a solution. Hence failure of the algorithm indicates that the original problem had no solutions. \( \square \)

Lemma 3 (Soundness and generality of type unification).

(a) If type unification succeeds with \( \Gamma \rightarrow \Delta \vdash \tau \equiv v \), then \( V_{TY}(\Gamma) = V_{TY}(\Delta) \), \( V_{GR}(\Gamma) \subseteq V_{GR}(\Delta) \) and it gives a most general solution \( \Gamma \sqsubseteq \Delta \vdash \tau \equiv v \).

(b) Correspondingly, if \( \Gamma | \Xi \rightarrow \Delta \vdash \alpha \equiv \tau \), then \( V_{TY}(\Gamma, \Xi) = V_{TY}(\Delta) \), \( V_{GR}(\Gamma) \subseteq V_{GR}(\Delta) \) and \( \Gamma, \Xi \sqsubseteq \Delta \vdash \alpha \equiv \tau \).

Proof. We proceed by induction on the structure of derivations, as discussed in the previous paper [4, Lemma 5]. There are four new rules:

For the Unit rule, the result follows from Lemma 1.

For Depends', the required property is identical to the inductive hypothesis.

For the Solve' and DependS'' rules, we use the Optimist’s lemma [4, Lemma 4], which states (more formally) that the minimal solution to a conjunction of problems is found by ‘optimistically’ solving the first problem in the original context, then solving the second problem in the resulting context. These rules fit the pattern as solutions to \( \alpha \equiv \tau(\vec{e}) \) are the same as solutions to \( \alpha \equiv_{TY} \tau(\vec{\beta}) \land \vec{\beta} \equiv_{GR} \vec{e} \) up to the equational theory.

Apart from the new rules, the argument for generality of the Repossess rule

is now more subtle, as group variables may appear in the context suffix \( \Xi \)
being moved into the previous locality. However, the invariant we established in Section 4.2 means that \( \Xi \) contains only type variables and unknown group variables that appear on their own in types. Any solution to the flex-rigid unification problem must move the entirety of \( \Xi \) past the marker, because all the group variables are genuine dependencies.

\[ \square \]

**Lemma 4 (Completeness of type unification).**

(a) If the types \( \upsilon \) and \( \tau \) are well-formed in \( \Gamma \) and there is some \( \theta : \Gamma \sqsubseteq \Theta \) such that \( \Theta \vdash \theta \upsilon \equiv \theta \tau \), then unification produces \( \Delta \) such that \( \Gamma \rightarrow \Delta \vdash \upsilon \equiv \tau \).

(b) Moreover, if \( \theta : \Gamma,\Xi \sqsubseteq \Theta \) is such that \( \Theta \vdash \theta \alpha \equiv \theta \tau \) and the following conditions are satisfied:

- \( \alpha \in V_{\text{TV}}(\Gamma) \), \( \tau \) is not a variable,
- \( \Gamma,\Xi \vdash \tau \) is TY, \( \Xi \) contains only type or group variable declarations
- \( \beta \in V_{\text{T}}(\Xi) \Rightarrow \beta \in FV_{\text{T}}(\tau,\Xi) \);

then there is some context \( \Delta \) such that \( \Gamma | \Xi \rightarrow \Delta \vdash \alpha \equiv \tau \).

**Proof.** First we establish that the system terminates, if viewed as an algorithm with inputs \( \Gamma \) (and \( \Xi \)), \( \upsilon \) (or \( \alpha \)) and \( \tau \), giving output \( \Delta \). The ‘unify’ judgments terminate because each recursive call removes a type variable from the context, decomposes the types or removes a group variable. The ‘solve’ judgments either shorten the whole context or the part of the context before the bar. Note that the Solve' and Depends'' rules may add group variables, but at least one type variable will be removed from the context before ExpandS calls ‘unify’ again. Only the Decompose rule makes more than one recursive call to type unification, and it decomposes types so it does not matter that the intermediate context may have more group variables.

Now we proceed by structural induction on the call graph, observing that each rule in turn preserves solutions, and that all (potentially solvable) cases are covered. The only cases not covered are rigid-rigid mismatches (e.g. unifying \( \upsilon \rightarrow \tau \) with \( \mathbb{F}(d) \)) and the flex-rigid problem \( \alpha \equiv \tau \) in context \( \Gamma,\alpha D | \Xi \) where \( \alpha \in FV_{\text{TV}}(\tau,\Xi) \). The latter has no solutions because the occur-check fails (if \( \alpha \) is in \( \Xi \) then the conditions of the lemma ensure \( \tau \) depends on it). For more details, see the previous paper [4, Lemma 7]. The algorithm may also fail in abelian group unification, for which completeness is by Lemma 2. 

\[ \square \]
An OCaml Library for Dynamic Typing

Julien Signoles

CEA LIST, Software Safety Laboratory
PC 94, F-91191 Gif-sur-Yvette Cedex, France,
Julien.Signoles@cea.fr

Abstract. This article presents a library providing a dynamic representation of OCaml types, including monomorphic instances of polymorphic types and abstract data types. That allows to consider types as first-class values. The library pragmatically fulfills the gap between OCaml and dynamic languages by mixing static and dynamic type checking. We focus on usage, implementation and properties of this library which comes within Frama-C, a Framework for Modular Analyses of C programs.

1 Introduction

“Well-typed programs do not go wrong” [18] is one of the main properties of ML-like static type systems as the one of Objective Caml (OCaml). Unfortunately this very welcome feature does not mix so well with usual programming features like unmarshalling. For instance, the OCaml standard library allows runtime errors while unmarshalling as shown by Figure 1.

bad_unmarshal.ml:
let () =
  (* saving an integer value in the file f.sav *)
  let cout = open_out "f.sav" in
  output_value cout 1;
  close_out cout;
  (* reading the integer value in the file f.sav *)
  let cin = open_in "f.sav" in
  let s : string = input_value cin in
  close_in cin;
  (* using it as a string *)
  print_endline s

$ ocamlc -o bad_unmarshal bad_unmarshal.ml
$ ./bad_unmarshal
Segmentation fault

Fig. 1. Runtime error while unmarshalling.

In such a case, dynamically-typed programming languages like Lisp and languages with introspection like Java check types at runtime and produces nice error messages instead of going wrong, which is a much better behaviour.

⋆ This work has been supported by the French ANR78U3CATZZ project U3CAT.
This article presents an OCaml library for dynamic typing, called Type, that provides an abstract data structure over types, including monomorphic instances of polymorphic types and abstract data types. That allows to consider OCaml types as first-class values. It pragmatically fulfills the gap between Objective Caml and dynamic languages: one part of type checking is done by the compiler thanks to the library API and the remainder is dynamically done by the library.

The library is embedded and used in Frama-C [7], an open source platform dedicated to source-code analysis of C software, which is an industrial project, both in its size (> 100 000 lines of OCaml code) and in its intended use for the certification, quality assurance, and reverse-engineering of industrial code [10]. Type is also compilable fully independently.

Outline. Section 2 shows main motivations for this work. Section 3 introduces the key points of its implementation. Section 4 explains useful extensions. Section 5 presents related works and compares them to our library.

2 Motivations

Frama-C uses the Type library in three different ways: to register and to access to APIs of dynamically-loaded plug-ins, to implement a feature called journalisation and to implement safe unmarshalling. Sections 2.1 to 2.3 present these features.

Throughout the article, we must also keep in mind the following key point: whatever solution we choose must be fully developed in OCaml, without any language extension nor dedicated library for which the Frama-C development team would not be able to provide long term support. The reason is Frama-C aims to be used in avionic programs which continue over decades: the tool and each external dependencies must be maintained during a whole avionic life-cycle. For instance, the use of a custom OCaml compiler or camlp4 is prohibited.

2.1 Dynamic API

Frama-C is an extensible platform where analysers are plug-ins which add functionalities to the Frama-C kernel [24]. Plug-ins may be dynamically loaded thanks to OCaml’s module Dynlink. That reduces Frama-C’s code size and postpones the choice of an effective implementation for a specific behaviour at runtime [6].

How accessing to plug-ins API? A module M can only dynamically access to a value of another module P through Dynlink if the API of P is statically available. Thus the interest of loading P dynamically is reduced. First it is not possible anymore that M uses an alternative implementation of P with an extended interface. Second it is not possible anymore to run M in the absence of P by deactivating some functionalities. Third it is not possible anymore to define mutually recursive plug-ins since linking forbids such compilation units.

The standard way that a module User accesses to a module Plugin dynamically loaded is thereby to use a third module Share in which Plugin registers its API. Figure 2 shows links between these modules while Figure 3 presents a
Fig. 2. The standard scheme of dynamic linkage in OCaml.

standard implementation which uses a global table `Share.dynamic_functions` in order to register the dynamic interfaces (only the function `f` in the example).

```
share.ml:
let dynamic_functions: (string, string -> unit) Hashtbl.t
 = Hashtbl.create 7
let get s = Hashtbl.find dynamic_functions s
let register s f = Hashtbl.add dynamic_functions s f

plugin.ml:
let f x = print_endline ("running f with " ^ x)
let () = Share.register "Plugin.f" f

user.ml:
let () = Dynlink.loadfile (Dynlink.adapt_filename "plugin.cmo")
let () = Share.get "Plugin.f" "s"
```

Fig. 3. Standard registration and access to a dynamic API in OCaml.

Unfortunately this standard implementation has a major drawback: the type \( \tau \) of each API function of each dynamically-loaded module must be known by module \texttt{Share} in advance. Even worse, one table by type must be defined (here the table `dynamic_functions` for values of type `string -> unit`): that does not scale up for a large application like \texttt{Frama-C}.

For solving this issue, one might use the alternative module \texttt{Share} shown in Figure 4. It solves previous issues by registering values of different types in an heterogeneous table which is implemented through type \texttt{Obj.t} and functions \texttt{Obj.repr} and \texttt{Obj.obj} of the OCaml standard library. The first function, of type \( \alpha \rightarrow \texttt{Obj.t} \), converts any OCaml value to a value of type \texttt{Obj.t} while the second one, of type \( \texttt{Obj.t} \rightarrow \alpha \), is its dual. Casts may be implemented by combining both functions, but they break type safety: continuing our example, the expression `Share.get "Plugin.f" 1` is well typed but leads to a runtime error. This solution is thereby worse than the previous one.

```
share.ml:
let dynamic_values: (string, Obj.t) Hashtbl.t = Hashtbl.create 7
let get s = Obj.obj (Hashtbl.find dynamic_values s)
let register s f = Hashtbl.add dynamic_values s (Obj.repr f)
```

Fig. 4. Unsafe-but-powerful registration and access to a dynamic API.
Our solution shown in Figure 5 solves drawbacks of previous solutions, but uses the same principle. It uses an heterogeneous table `dynamic_values` indexed

```
share.ml:
let dynamic_values: Type.String_tbl.t = Type.String_tbl.create 7
let get s ty = Type.String_tbl.find dynamic_values s ty
let register s ty f = Type.String_tbl.add dynamic_values s ty f
```

```
plugin.ml:
let f x = print_endline ("running f with" ^ x)
let () =
  Share.register "Plugin.f"
  (Datatype.func Datatype.string Datatype.unit) f
```

```
user.ml:
let () = Dynlink.loadfile (Dynlink.adapt_filename "plugin.cmo")
let () =
  Share.get "Plugin.f"
  (Datatype.func Datatype.string Datatype.unit) "s"
```

Fig. 5. Registration and access to a dynamic API through the `Type` library.

by strings and provided by the `Type` library. Heterogeneity solves drawbacks of the standard solution of Figure 3 in the same way than the solution of Figure 4. But safety is now ensured thanks to a so-called `type-value` which must be provided as argument to `Share.register` and `Share.get`. Here the type-value is `Datatype.func Datatype.string Datatype.unit` which corresponds to the static type `string -> unit`. Section 3 provides further details about safety.

### 2.2 Journalisation

Journalisation is a way to generate an OCaml script (called `journal`) which replays each user actions, in particular when they are done through a GUI. The interest is threefold. First, debugging is easier in case of unexpected errors. Second it may be used as a macro language for automating sequences of user actions. Third it helps developers (especially beginners) for quick prototyping.

In Frama-C, the developer must register each function `f` to be written in the journal whenever `f` is totally applied. For this purpose, a safe heterogeneous table provided by the `Type` library is used. The internal algorithm of journalisation also uses introspection facilities provided by the library: it takes a function `f` and its dynamic type `τ` as arguments and builds by induction on the number of arrows in `τ` a new function observationally equivalent to `f`, but writing each of its calls in the journal whenever it is totally applied.

\[^1\] Details of this algorithm is beyond the scope of this article.
2.3 Unmarshalling

As previously shown in Figure 1, unmarshalling is not safe in OCaml. If unmarshalling a few values by using a simple protocol is easy enough to be implemented without error, it is not the case in Frama-C. Indeed Frama-C massively uses hash-consing [9] which is a programming technique requiring to modify loaded values when unmarshalling in order to ensure maximal sharing (the key property of hashconsing) between values already in RAM and values loaded from disk.

For this purpose, Frama-C implements its own algorithm for unmarshalling which allows to apply specific operations on loaded values, according to its static type\(^2\). This algorithm is fully written in OCaml, but intensively uses the unsafe module Obj even in its interface. For instance, modifying a value of a type \(\tau\) requires to implement a function of type \(\text{Obj}.t \rightarrow \text{Obj}.t\). Thanks to the Type library, this algorithm has a safe front-end in which this function must have type \(\tau \rightarrow \tau\). Furthermore, the global unmarshalling function has now the type \(\text{in-channel} \rightarrow \alpha\text{Type}.t \rightarrow \alpha\) ensuring safety: in case of error, either the OCaml compiler detects that the type of the type-value does not match the type of the context, or the Type library detects that the type-value written on the disk when serialising does not match the type-value provided as argument.

3 Implementing the library

This section explains how the library works. First Section 3.1 shows how to create type-values. Next Section 3.2 provides implementation details of heterogeneous tables. Finally Section 3.3 presents the properties guaranteed by the library.

3.1 Type-Values

The library Type defines an abstract polymorphic type \(\alpha\text{Type}.t\) which represents type-values and a unique constructor \text{register}. Figure 6 shows them together with two projectors \text{name} and \text{repr} and one equality function \text{equal}.

Let us define for any monomorphic type \(\tau\), \(\mathcal{T}_\tau\) the sets of \(\tau\)-type-values:

\[
\mathcal{T}_\tau \triangleq \{\text{type-value } ty \mid ty : \tau \text{ Type}.t\}.
\]

With this definition, it is now possible to establish the key property of the library saying that a value with type \(\tau\text{ Type}.t\) represents the static type \(\tau\).

Property 1 (Injection from monomorphic types to type-values) equality over type-values implies equality between corresponding monomorphic static types, that is:

\[
\forall \text{ monomorphic types } \tau \text{ and } \tau', \forall ty \in \mathcal{T}_\tau, \forall ty' \in \mathcal{T}_{\tau'}, \quad \text{Type}.\text{equal } ty ty' \implies \tau = \tau'.
\]

\(^2\) Like journalisation, details of this algorithm is beyond the scope of this article.
This property is based on the fact that the string given as first argument of the function Type.register is used as an unique key: each new type-value is different from each other existing ones. Furthermore, the second argument of the function Type.register, the so-called representative value of the type, statically links the new registered type-value and the represented static type thanks to the type variable α of the phantom type \([\alpha] \text{Type.t}\): this type variable is indeed instantiated with the type of the representative value.

This link is between a representative value \(r\) and its static type \(\tau\), but not between \(r\) and the physical representation of \(\tau\). For instance, the representative value of type \(\text{int}\) cannot be the same than the one of type \(\text{bool}\), even if both types have the same physical representation. This remark is important for interprocess communications: it is not possible to write an integer on a channel and to read it as a boolean.

Type.register provides an easy way to build type-value for any monomorphic type, what is expressed by property 2. The Type library offers a module Datatype which provides a type-value for each OCaml basic type. For instance, the implementation of float-type-value of type float Type.t is as follows.

\[
\text{let float} = \text{Type.register "float" 0.0}
\]

**Property 2 (Any set \(\mathcal{T}_\tau\) is inhabited)** For any monomorphic type \(\tau\), it is possible to build a type-value belonging to \(\mathcal{T}_\tau\).

However it is theoretically possible to build two different type-values which represent the same static type. For instance, the value float2 below represents the static type float in the same way than the previous value float.
It is very easy to enforce naming convention avoiding this issue. That is what is done in Frama-C. Thus, in practice, it is reasonable to expect that the reciprocal of property 1 holds.

**Hypothèse 1 (Bijection between any monomorphic type $\tau$ and $T_\tau$)** The equality $\text{Type.equal}$ over type-values coincides with the equality over monomorphic types, that is:

$$\forall \text{ monomorphic types } \tau \text{ and } \tau', \forall \text{ty} \in T_\tau, \forall \text{ty}' \in T_{\tau'}, \tau = \tau' \iff \text{Type.equal ty ty'}.$$ 

Actually this hypothesis (valid in practice) is only required to establish completeness (property 4), but not correctness (property 3), the most important one: if a type has sadly two different representatives (breaking hypothesis 1, but not property 1), the only potential issue is that correct programs might not type check. Still no program going wrong at runtime would be accepted.

The Type library requires so far to register each type we want to use. That may seem quite annoying, but it is not. First each type of the OCaml standard library are pre-registered in the module Datatype. Second registering a new type-value is straightforward: only two simple arguments are required. Third, in practice, the registration is done through module Datatype which provides additional functionalities at the same time (such as pretty printers).

### 3.2 Safe-to-use Heterogeneous Tables

Implementing heterogeneous table in OCaml in a safe way is not possible. The standard solution, even if very dangerous, is the one already shown in Figure 4. Our implantation of module String_tbl, previously used in Figure 5, is very similar to this unsafe version, but with a fundamental difference: it relies on a safe interface using type-values which guarantees that any bad use of this module cannot lead to a runtime error. This interface is shown Figure 7. It provides similar functions to hash-tables, but functions add and find take a type-value $t$ as additional argument used as guard: $t$ must represent the same type as the one of the added/searched value. These additional guards ensure safety by doing checks at three different times.

```ocaml
module String_tbl : sig
    type t
    val create : int -> t
    val add : t -> string -> $\alpha$ ty -> $\alpha$ -> unit
    val find : t -> string -> $\alpha$ ty -> $\alpha$
end

Fig. 7. Interface of heterogeneous tables.
```

First, the type of the function add is used by the OCaml compiler to ensure that the type-value $t$ given as argument represents the same type as the type of
the value \( v \) added in the table. In the following example, the first statement is well-typed while the second one is not.

```ocaml
let () = Type.String_tbl.add t "key" Datatype.string "toto"
let () = Type.String_tbl.add t "key" Datatype.string 1
```

Next the type of function `find` is used by the OCaml compiler to automatically infer the type of the returned value according to the type of the type-value given as argument and to verify that it is usable in the context. In the following example, the first statement is well-typed while the second one is not.

```ocaml
let () = print_string (Type.String_tbl.find t "key" Datatype.string)
let () = print_int (Type.String_tbl.find t "key" Datatype.string)
```

Finally the implementation of module `String_tbl` shown in Figure 8 dynamically verifies that type-values given as argument to functions `add` and `find` are equal. In the following example, the first lookup runs without error while the second one raises a `Type.String_tbl.Incompatible_type` exception with the message “key has type string but is used with type int.”

```ocaml
let () = Type.String_tbl.add t "key" Datatype.string "toto"
let () = print_string (Type.String_tbl.find t "key" Datatype.string)
let () = print_int (Type.String_tbl.find t "key" Datatype.string)
```

```ocaml
module String_tbl = struct
  type dynamic = { ty: string; o: Obj.t }
  type t = (string, dynamic) Hashtbl.t
  exception Already_exists of string
  exception Unbound_value of string
  exception Incompatible_type of string
  let type_error s ty ty' =
    raise (Incompatible_type
      (Format.sprintf "%s has type %s but is used with type %s." s (name ty') (name ty)))
  let create = Hashtbl.create
  let add tbl s ty x =
    if Hashtbl.mem tbl s then raise (Already_exists s);
    Hashtbl.add tbl s { ty = ty; o = Obj.repr x }
  let find tbl s ty =
    try
      let data = Hashtbl.find tbl s in
      (* dynamic type checking *)
      if ty <> data.ty then type_error s ty data.ty;
      Obj.obj data.o
    with Not_found → raise (Unbound_value s)
end
```

Fig. 8. Implementation of heterogeneous tables.
Actually the library provides generic functors implementing different heterogeneous tables. The module Type.String_tbl is only one particular instance.

3.3 Properties

Using unsafe coercions through functions Obj.obj and Obj.repr may break the usual strong correctness theorem [18] of the OCaml type system. However, in the context of our library, the use of these functions is fully safe, as expressed by the following property.

Property 3 (Strong correctness) Excepting unsafe expressions used elsewhere outside module Type, well-typed expressions do not go wrong.

Furthermore, assuming hypothesis 1, any use of these safe heterogeneous tables does not introduce unwelcome restrictions. That is expressed by the two properties below.

Property 4 (Completeness) Assuming hypothesis 1, for any program \( p \) which does not go wrong and uses unsafe heterogeneous tables (as shown in Figure 4), the program \( p \) where occurrences of such tables are replaced by similar occurrences of Type.StringTbl is well-typed, does not go wrong and does not raise any exception coming from the Type library.

Property 5 (Semantic equivalence) For any program \( p \) which does not go wrong, uses unsafe heterogeneous tables (as shown in Figure 4) and is evaluated to a value \( v \), if the program \( p \) where occurrences of such tables are replaced by similar occurrences of the module Type.StringTbl is well-typed and does neither go wrong nor raise any exception coming from the Type library, then it is also evaluated to \( v \).

4 Extensions

Section 4.1 extends the library to monomorphic instances of polymorphic types, while Section 4.2 extends it to abstract data types.

4.1 Monomorphic Instances of Polymorphic Types

The library would be unsafe if the function register returned a polymorphic value instead of a monomorphic one, because the type-value would not represent a unique type anymore but the set of monomorphic instances of this type: that breaks property 1. For instance, assume that it is possible to build a type-value \( ty \) of type \( \alpha \) list Type.t. Let \( t \) be an heterogeneous table of type Type.StringTbl.t. It is possible to write the following code.

```ocaml
let () = Type.StringTbl.add t "f" ty [ 1 ]
let () = print_string (List.head (Type.StringTbl.find t "f" ty))
```
This program is well-typed since the type variable \( \alpha \) is unified with type \( \text{int} \) when \([1]\) is added into \( t \) and then unified with type \( \text{string} \) when "f" is searched in \( t \). Furthermore, the Type library does not raise any exception since the type-value is the same when adding and when searching the value.

Here, the type variable \( \alpha \) of the type of \( ty \) is unified differently when used in functions \( \text{add} \) (unified with \( \text{int} \)) and \( \text{find} \) (unified with \( \text{string} \)). Thus this code leads to a runtime error. That is why all the properties only hold for monomorphic types. Fortunately it is actually not possible to build polymorphic types values: the type variable \( \alpha \) of the function \( \text{Type.register} \) is not generalisable since the second argument \( \text{repr} \) of type \( \alpha \) is added in an imperative hash table \([15]\). That is a quite unusual case where impurity ensures safety.

Thus, even if the Type library does not handle full polymorphism, the use of polymorphic values does not break its safety. Furthermore, the other solutions previously shown Figure 3 and 4 also lead to the same issue as they use hash-tables. However monomorphic instances of polymorphic types may still be registered as follows.

```plaintext
let int_list = register "int list" [0]
```

But such a solution is not tractable in practice, since it becomes impossible to pre-register in the library all the instances. Furthermore, if a plug-in registers type \( \text{bool list} \) for its own purpose, no other plug-in can do it (except if it registers it with a different name, thus breaking hypothesis 1).

For solving this issue, module \( \text{Type} \) allows to register each polymorphic type \( \text{via} \) a functor \( \text{Polymorphic} \) shown in Figure 9. The functor body contains a function \( \text{instantiate} \) which generates a type-value by monomorphic instantiation of the registered polymorphic type. This function uses memoisation \([17]\) in order to prevent multiple type-values by instance.

```plaintext
module Polymorphic(T:sig
  type \( \alpha t \) (* registered polymorphic type *)
  val repr: \( \alpha \rightarrow \alpha t \) (* how to create a representative for it? *)
  val name: string \rightarrow\ string (* how to create a name for it? *)
end) : sig val instantiate: \( \alpha t \rightarrow\ \alpha T.t \end =
```

```plaintext
struct
  let instances : (string, string) Hashtbl.t = Hashtbl.create 17
  let instantiate ty =
    try Hashtbl.find instances ty
    with Not_found →
    let repr = register (T.name (name ty)) (T.repr (repr ty)) in
    Hashtbl.add instances ty repr;
   repr
end
```

**Fig. 9.** Functorial approach for polymorphism.

As example, it is shown below how to apply this functor for being able to generate monomorphic instances of lists: applying \( \text{list int} \) returns a type-value \( v \) for \( \text{int list} \). Applying \( \text{list int} \) afterwards returns \( v \) again.

45
module List = Polymorphic
(struct
  type 'a t = 'a list
  let name s = "(" ^ s ^ ") list"
  let repr ty = [ ty ]
end)
let list = List.instantiate

In practice, the library provides modules for polymorphic types of the OCaml standard library. Furthermore it implements pretty printers of type names in order to provide readable error messages.

The only drawback of this approach is that it requires one different functor by number of free type variables in the polymorphic types: the functor Polymorphic only handles types of the form \( \alpha t \) (one type variable), the library provides another functor Polymorphic2 for handling types of the form \( (\alpha, \beta) t \) (two type variables), but it does not provide functor for types with more than three free type variables. For instance, the value Datatype.func used in Figure 5 is equivalent to Function.instantiate generating type values for monomorphic instances of arrow types \( \alpha \to \beta \).

### 4.2 Abstract Data Type

Consider the following plug-in which manipulates singleton lists.

type t = int list
let single n = [ n ]
let head l = match l with [ x ] -> x | [] | _ :: _ :: _ -> assert false

Providing the following interface is enough to implement an abstract data type (ADT) ensuring that function head never fails since only list with a single element can be built with type \( t \).

type t
val single : int -> t (* constructor *)
val head : t -> int (* observer *)

However, if this module is dynamically loaded, no value outside the plug-in can use this abstract type since it is not statically known as previously shown by Figure 2. It is yet possible to create a type-value \( ty \) for this type.

share.ml: (* same as Figure 5 *)
plugin.ml:
(* continuing code at beginning of this section *)
let ty = Type.register ~abstract:true "Plugin.t" [ 0 ]
let () =
  Share.register "Plugin.single" (Datatype.func Datatype.int ty) single;
  Share.register "Plugin.head" (Datatype.func ty Datatype.int) head

However \( ty \) is not visible outside Plugin. Thus no user is able to call functions Plugin.single and Plugin.head. It is still possible to export both functions by
using Datatype.list Datatype.int instead of ty, but that breaks abstraction in a way that the code assertion does not hold anymore.

In order to solve this issue, it is possible to flag the newly registered type-value with abstract:true. Thus it is now possible to access it dynamically via its name through the function Type.get of type string→abstract Type.t. Here abstract is actually an empty type provided by the Type library. All type-values representing different static types and accessed through Type.get share the same type abstract Type.t. That breaks property 1. To preserve safety, the library performs additional type checks when it encounters a value of type abstract Type.t. For instance, here is how we can now access to the previously registered type-value ty.

```
user.ml:
let ty = Type.get "Plugin.t"
let 1 = Share.get "Plugin.single" (Datatype.func Datatype.int ty) 1
let () =
  print_int (Share.get "Plugin.head" (Datatype.func ty Datatype.int) l)
```

As the type of ty is abstract Type.t, the type of the value 1 is abstract. There is no function taking a value of type abstract as argument, thus the only effective way to use 1 is to give it as argument of a dynamically registered function waiting a value of type abstract Type.t. That is the case of

```
f = get "Plugin.head" ty Datatype.int. When applying f, the Type library verifies that the argument of type ty has been previously built by a function returning a value of type Plugin.t corresponding to the type-value ty. When it is not the case, an exception is raised with the message "argument 1 of Plugin.head not built with a constructor of type Plugin.t".
```

In order to implement this behaviour, the function Type.String_tbl.add adds in the table a value v' observationally equivalent to v, but registering each value built from an ADT constructor (a function f returning a type corresponding to a type-value ty flagged as abstract) by using the following skeleton.

```
fun x →
  let y = f x in
  register_value ty y;
  y
```

Similarly, each ADT observer g checks that its argument of a type corresponding to a type-value ty flagged as abstract was previously built by a constructor of the same ADT by using the following skeleton.

```
fun x → if not (mem_value ty x) then fail; g x
```

Figure 10 gives the full algorithm. It uses introspection functions provided by our library in order to detect if a type-value represents an arrow type τ₁→τ₂ (function Function.is_instance_of) and then to get τ₁ and τ₂ (function Function.get_instance). Module Abstract_values is omitted but keeps a pointer on each values of an ADT. It uses weak tables [9] to prevent memory leak. This algorithm uses the evil function Obj.magic, of type α→β which allows to change arbitrarily the type of any value.
let rec abstract s n ty x : Obj.t =
  if is_abstract ty then Abstract_values.add ty x;
  if Function.is_instance_of ty then
    let ty1, ty2 = Function.get_instance ty in
    (* ok: [x] is a function here *)
    let f : α → β = Obj.magic x in
    Obj.repr (fun (y: α) →
      if is_abstract ty1 && not (Abstract_values.mem a y) then
        fail; (* raises an exception with a nice error message *)
        (Obj.obj (abstract s (succ n) ty2 (f y)) : β))
    else
      Obj.repr x

let add tbl s ty x =
  if Hashtbl.mem tbl s then raise (Already_exists s);
  Hashtbl.add tbl s { ty = ty; o = abstract s 1 ty x }

Fig. 10. Additional checks for ADT.

5 Related Work

Mixing dynamic and static typing was already quite widely studied, especially for functional programming languages. First dynamically-typed programming languages can be extended with static type annotations in order to perform partial static type checking known as gradual typing [23]. Here we do the opposite by introducing runtime type checks in a statically-typed programming language.

In this area, most works are language-based approaches: either a fully new language is designed or an existing one like Haskell or ML is extended with additional features. In both cases, the goal is to embed some dynamic typing facilities in the compiler. Thus these approaches, even of interest, can not be used in our context since they require to use custom compilers and/or language extensions. Nevertheless they are comparable with our approach.

Most of them use dynamics, introduced by Cardelli in the Amber programming language [5]. A dynamic is a pair of a value $v$ and a type expression $\tau$, such that value $v$ has type $\tau$. As shown in Figure 8 by our type dynamic, it is easy to encode monomorphic dynamics with our library. A `typecase` primitive is usually provided in order to choose at runtime the action to be executed according to the static type. Our type-value equality and functions `is_instance_of` allow to encode this feature in a clumsy way, but this can be fixed by using Camlp4 if allowed. Extending ML with dynamics was studies in the 90’s without [1] and with [2, 16] polymorphism. The work of Leroy and Mauny [16] also deals with safe marshalling and was implemented in a very old CAML, but dynamics do not exist anymore in OCaml today. More recent functional languages, like Alice ML [21], Clean [19] or GML [13] now provide dynamics.

Some other works are not directly related to dynamics. Thereby Acute [22] and its extension HashCaml [4] implement safe marshalling by using type representations computed from hashes of module definitions. Henry and al study an extension of OCaml for providing safe unmarshalling without modifying the OCaml runtime [14]. More theoretical framework have also been studied. For
instance, Crary and al [8] studied a lambda-calculus allowing to represent types by terms in order to select code at runtime according to a static type. However they do not study imperative features and it does not seem easy to add them to their calculus. In all the above works, some of them [4, 13, 14, 22] cannot be applied in all our case studies, while only few [4, 21] take care of abstraction.

Even if language-based approaches are more largely studied, some works are library-based approaches. First Haskell’s compilers Hugs and GHC provide dynamics as a library. They use unsafe coercions and only support monomorphic instances of polymorphic types like our library. However they do not take care of abstraction. A fully safe version of these Haskell libraries was also proposed [3], but is is not typable in OCaml since it requires existentially and universally quantified types and constructors.

ML-like languages do not have any dynamic library by default. The Deriving extension of OCaml [28] provides a preprocessor and libraries for type-safe marshalling and dynamic typing. It is similar to our library on some points: there is for instance combinators like our functions Datatype.List and Datatype.func in order to handle monomorphic instances of polymorphic types. But it is fully based on Camlp4 extensions, while our library is in pure OCaml. In the same way, Dynaml [11] and Dyn [25] provides dynamics as a library by using Camlp4 extensions. There are also known safe encodings of universal types in ML [26, 27] in which any value of any type may be injected to and surjected from. Such types allow to safely define heterogeneous structures. But they cannot be used for dynamic linkage since injector and surjector must be defined simultaneously, while Section 2 has shown that the injector has to be defined in module Plugin and the surjector in module User. The Coq proof assistant also contains an unsafe implementation of universal types through its module Dyn.

Another close work of Type is done at LexiFi, a provider of software applications and infrastructure technology for the capital markets industry. LexiFi defines a type α.ttype which is a runtime representation of monomorphic types [12]. As for us, it is a phantom type used for safety purpose in order to dynamically represent static monomorphic types, but it does not preserve abstraction. Furthermore our goals are different: LexiFi uses the runtime representation of types in order to automatically generate parts of graphical user interfaces, to describe SQL schemes, for inter-process communications and for debugging. Another difference is that values of type α.ttype are automatically generated by a custom version of the OCaml compiler.

6 Conclusion

This article presented the Type library which provides a dynamic representation of OCaml types, including monomorphic instances of polymorphic types and abstract data types. That allows to consider OCaml types as first-class values. It

---

3 Consistently with the name Dyn of this module, its documentation uses the word “dynamics” instead of “universal types”. However the code actually provides an unsafe implementation of universal types.
pragmatically fulfills the gap between OCaml and dynamic languages by mixing static and dynamic type checking.

This library was designed for and is used in Frama-C, a platform for source-code analysis of C software, in three different ways: to register and to access values of dynamically-loaded code, to add safety to an efficient algorithm dedicated to unmarshalling in presence of hashconsing, and to implement an algorithm of journalisation which generates OCaml code from user actions. The library ensures safety and provides introspection facilities (even limited). Only the key points of the library were presented here: the implementation is around 2400 lines of code and provides additional functionalities like handling labels and optional arguments.

Some potentially useful features are not yet implemented in the library. The most important one consists in registering open types like polymorphic types, polymorphic variants or objects. Beyond the difficulties to implement these features, none of them are required in Frama-C today, or even planed to be required, in a place where dynamic typing is used.

Acknowledgements

We are grateful to B. Blanc, L. Correnson, Z. Dargaye, A. Frisch, P. Herrmann and B. Monate for fruitful discussions about this article. J.-D. Hatzenbühler and J. Peeters participated to a preliminary implementation of the library as well as its usage in Frama-C.

References

12. A. Frisch. Personal communication by e-mail, September 2009.
Abstract. In this paper we show that a prototypic subtype relation that can neither be defined as a least fixed point nor as a greatest fixed point can nevertheless be defined in a dependently typed language with inductive and coinductive types and an impredicative universe of propositions. The definition proceeds alike a fold in functional programming, although a rather unusual one, that is not applied to any starting object. There has been a similar construction in Coq by Nakata and Uustalu recently, however, our case is not concerned with bisimilarity but a weaker notion of similarity that corresponds to recursive subtyping.

1 Introduction

It is common in practice to have datatypes formed by nested least and greatest fixed points. For example, consider a grammar, and parse trees of derivations in that grammar that are allowed to be infinite only below certain non-terminal nodes. Or, a semantic model of a programming language where we distinguish between termination and diverging computation. With dependent types, it is possible to define types such as of grammar or parse tree. However, it is not straightforward to define nested fixed points using implementations of inductive and coinductive type definitions. This is mainly because these type definitions are subject to strong syntactic checks in current implementations of dependently typed languages. There are at least two different methods to encode nested fixed points in type theory that are both known as mixed induction-coinduction, the first is defined in [1] and the second, in [2]. The former uses a programming construct of suspension computation monad, while the latter seems to rely on a variant of fold function. Suspension monad is efficient and intuitive, however, it has to be supported by the programming language rather than simply implemented on top of it, for which many dependently typed provers would require substantial re-engineering. Not having a sufficient resource for rewriting the implementation of a prover, we choose the second, probably not so efficient but maybe a bit more portable method and apply the fold pattern on top of the language.

The language in question is Coq [3]. It has dependent products of the form \( \forall (x : A). B \) where \( x \) is a variable which is bound in \( B \); the case when \( x \) is not
free in $B$ is denoted $A \rightarrow B$ which is a simple, non-dependent type. Also, Coq
features inductive and coinductive type definitions. For the sake of presentation,
we do not provide listings of Coq code, which would be plain ASCII. Instead,
throughout the paper, we use a human-oriented type-theoretic notation, where $\star$
denotes the universe of types (which is predicative, $\text{Type}$), $\star'$ denotes the universe
of propositions (which is impredicative, $\text{Prop}$), and inductive and coinductive
definitions are displayed in natural-deduction style with single and, respectively,
double lines.

**Contribution.** In this paper we show that a prototypic subtype relation that
can neither be defined as a least fixed point nor as a greatest fixed point can
nevertheless be defined in a dependently typed language with inductive and
coinductive types and an impredicative universe of propositions. The definition
proceeds alike a fold in functional programming, although a rather unusual one,
that is not applied to any starting object. There has been a similar construction
in Coq by Nakata and Uustalu recently [2], however, our case is not concerned
with bisimilarity but a weaker notion of similarity that corresponds to recursive
subtyping.

The motivation for this work is better understanding of termination issues
in subtyping, including formalisms such as extended regular expressions, and
paving the way for further extensions and provably correct practical applications.

**Outline.** In Sec. 2 we explain the subtyping relation construction method. In
Sec. 3 we define the object language of recursive types formally, using Coq as
the meta-language. In Sec. 4 we define subtyping in the meta-language. Sec. 5
contains the statement of soundness and completeness of recursive types with
respect to finite and infinite trees, and a description of a useful approach to
decidability of subtyping in a dependently typed language. The powerful method
of monadic substitution is described in Sec. 6. The alter ego mixed induction-
coinduction method is described in Sec. 7. Finally, in Sec. 8 we give concluding
remarks.

## 2 The fold pattern

Here is the polymorphic type of the familiar list-based left fold function:

\[
\text{foldl} : \forall (S T : \star) (f : T \rightarrow S \rightarrow T) (a : T) (l : \text{list } S). T
\]

(On a fundamental approach to fold, the reader is advised to refer to [4].) Application
of foldl to an appropriately typed function $f$, an object $a$ of target type
and a list $l$ of objects of source type yields an object of target type $T$. The result
of iterative computation of $f$ on the list $l$ starting from $a$ is aggregated on the
left.

Let us now drop the requirement that fold starts from some object. This
removes the first argument of the function $f$ altogether. Our hypothetic fold has
two dependent arguments: a function $f$ and a collection $l$ (which is not quite a
list) of objects of $S$. This is in fact a description for the definition of the following function type by coinduction:

$$\begin{align*}
f : (\forall E F. R E F \rightarrow E \leq F) \\
l : E \leq R F
\end{align*}$$

where $E$ and $F$ are polymorphic arguments. From its type, we can see that $\leq$-intro has two dependent arguments, $f$ and $l$, and yields an object (in fact, a proof) inhabiting a particular case of relation. The function $\leq$-intro is the constructor of the coinductive relation $\leq$. The function $f$ can indeed be seen as a mapping of a proof that from an object $E$ we can access another object $F$ by the relation $R$ to a corresponding proof that from $E$ we can also reach $F$ by $\leq$. In other words, $f$ is a coercion from $R$ to $\leq$. The interpretation of the argument $l$ is a bit more involved. Think of $\leq_R$ as a finite relation encapsulating another, infinite one in such a way that an infinite number of steps is possible only finitely. The latter sounds rather speculative, however, the intuition is that $\leq_R$ is alike a type of finite list of certain abstract, possibly infinite objects. The codomain of $\leq$-intro is a type of infinite object in which we collapse all the infinite components of the argument $l$. The object in the codomain is coinductive, and it is only thanks to the premisses of $\leq$-intro that we are able to compare elements of pairs in the domain of $\leq$ in a finite number of steps possibly infinitely.

It is worth noting that having a coinductive type definition such as that of $\leq$ is nothing close to requiring an infinite amount of memory for objects of that type. The shape of such an object is a regular tree which may have an infinite unfolding but in itself is a finitely presented entity.

### 3 Recursive types

Below in this section we give a proper inductive definition of recursive types, our object language, in Coq as the the meta-language. However, first recall a traditional definition of the set of recursive types that uses a grammar [5, 6]:

$$E, F ::= \bot | \top | X | E \rightarrow F | \mu X. E \rightarrow F$$

where $X$ is a symbolic variable taken from a set of variables. The least fixed point operator $\mu$ binds free occurrences of the variable $X$ in $E \rightarrow F$. This definition neither has products or sums that are needed for practical programming. However, we do not consider product or sum types in this schematic implementation because their treatment is alike that of $\rightarrow$, see, e.g., [6]. Moreover, we choose to replace named variables in the definition by nameless de Bruijn variables, which yields an equivalent and yet more tangible construction.

Now, our working definition of recursive types is by induction as follows:

$$\begin{align*}
\text{ty} : \mathbb{N} \rightarrow \star \\
\bot : \text{ty} n \\
\top : \text{ty} n \\
i : \mathcal{I}_n \\
i X_i : \text{ty} n \\
E : \text{ty} (1 + n) \\
F : \text{ty} (1 + n) \\
\mu E \rightarrow F : \text{ty} n
\end{align*}$$
where the constructors are, respectively, the empty type \( \bot \), the unit type \( \top \), a variable, the function type constructor \( \rightarrow \) and the least fixed point arrow type constructor \( \mu \). The dependent type \( I_n \) represents the first \( n \) natural numbers, and therefore an object \( i \) of type \( I_n \) is a pair consisting of a natural number \( m \) and a proof of \( m < n \).

Recursive types have a correspondence with non-wellfounded (finite or infinite) trees with the following definition by coinduction:

\[
\begin{align*}
\text{tree} & : \mathbb{N} \to * \\
\bot & : \text{tree} \\
\top & : \text{tree} \\
i : I_n & \\
X^\infty_i & : \text{tree} \\
t & : \text{tree} \\
u & : \text{tree}
\end{align*}
\]

Intuitively, trees are views of \( \mu \)-types unfolded \textit{ad infinitum}. We denote the tree corresponding to a type \( E \) by \([E]\). The straightforward subtree relation \( \text{tle} \) on \( \text{tree} \) is denoted by \( \leq^\infty \) (omitting the implicit argument \( n \)):

\[
\begin{align*}
\bot & \leq^\infty t \\
t & \leq^\infty \top \\
i : I_n & \\
u_1 & \leq^\infty u_1 \\
u_2 & \leq^\infty u_2
\end{align*}
\]

Thus, two recursive types are in the subtype relation when their potentially infinite unfoldings are in the subtree relation. Traditionally, subtyping theorems are stated in terms of inductive limits of sequences of approximations of unfoldings of recursive types (e.g., in [6]). Instead of using explicit induction in that way, we rather rely on dependent types of the CIC which allow to define a powerful monadic structure encapsulating unfolding \textit{ad infinitum}. The point here, similar to a remark made by Amadio and Cardelli in [6], is that unfoldings of recursive types are \textit{regular} trees, which we treat using a mix of induction and coinduction.

4 Definition of recursive subtyping

We define the weak similarity relation \( \text{tyle} \) by folding the inductive part of the definition into the coinductive one. Our technique is an illustration of a generic method for folding one relation in another. We use the impredicative universe of propositions that we denote by \( \star' \), which is needed for the proof of soundness and completeness. First, we define the inductive part \( \text{tyle} \) of the subtyping relation (denoting \( \text{tyle} \) by \( E \leq_R F \), suppressing the implicit argument \( n \)):

\[
\begin{align*}
\text{tyle} & : \forall n. (\text{ty} n \to \text{ty} n \to \star') \to \text{ty} n \to \text{ty} n \to \star'
\end{align*}
\]

\[
\begin{align*}
\bot & \leq_R E \\
E & \leq_R \top \\
R E F & \leq_R R G H \\
F \rightarrow G & \leq_R E \rightarrow H \\
\mu E \rightarrow F & \leq_R \text{unfld} E F \\
E & \leq_R F \\
F & \leq_R G \\
E & \leq_R G
\end{align*}
\]

55
where \texttt{unfld} is the operation that unfolds a \(\mu\)-redex by substituting the variable 0 in the term \(E \to F\) with the term \(\mu E \to F\). This operation is defined in Sec. 6. Having the rules for reflexivity and transitivity in the inductive part of the subtype relation is essential for this construction. Indeed, by having these rules explicitly, we are able to compare elements of pairs in the domain of the subtype relation in a \textit{finite} number of steps possibly \textit{ininitely}. Leaving transitivity out of the definition would collapse finite and infinite transitivity chains to infinite ones only.

Next step is to fold the inductive relation and produce a weak similarity. This is done by the single-constructor coinductive type:

\[
\text{tyle} \ n \ : \ \text{ty} \ n \to \text{ty} \ n \to \star'
\]

We denote \(\text{tyle} \ n \ E \ F\) by \(E \leq F\).

\[
\forall \ E \ F. \ R \ E \ F \to E \leq F \quad E \leq R \ F \Rightarrow E \leq F
\]

The only introduction rule for \(\leq\) has two hypotheses, namely, that \(R\) is a subrelation of \(\leq\), and that \(F\) is \(\leq_R\)-accessible from \(E\) in finitely many steps (since \(\leq_R\) is an inductive relation).

5 Soundness and completeness

Main Theorem (Soundness and completeness).

\[
\forall \ (n : \mathbb{N}) \ (E \ F : \text{ty} \ n). \ E \leq F \leftrightarrow [E] \leq^\infty [F]
\]

The “only if” direction (completeness) follows by a straightforward application of the coinduction principle. For the “if” direction (soundness), we define the weak head normal form of the relation \(\leq^\infty\) and solve the problem via this notion, which is a common workaround helping to ensure syntactic guardedness of the proof [1, 2].

The soundness and completeness result allows us to tell that our definition of syntactic subtyping is correct with respect to the tree semantics. In a proof assistant this is only a change in representation. Besides, both representations are propositional. Since proof search is undecidable on the universe of propositions in general, it is impractical, and likely impossible, to use either of the representations for efficient proof search in a prover. Instead, we can use the approach which is known as the \textit{two-level approach} [8] or \textit{small-scale reflection} [9].

We can implement a decision procedure for a class of propositional goals \(G \in \star'\) by

1. first defining a type of \textit{codes} \ \texttt{goal} : \star and an interpretation function \([.\.]: \texttt{goal} \to \star'\) surjective on \(G\),
2. and then defining a decision algorithm \( \text{dec} : \text{goal} \rightarrow \text{bool} \) which can be proved sound and complete with respect to the propositional interpretation, that is,

\[
\forall g : \text{goal}. \, \text{dec } g = \text{true} \leftrightarrow [g]
\]

As a result, to prove \( P \in G \), it is sufficient to compute \( \text{dec } g \), where \( g \) is the code for \( P \).

Alternatively, soundness and completeness of the decision algorithms is an object of inductive type \( \text{decidable} : \star' \rightarrow \text{bool} \rightarrow \star \) defined by

\[
\begin{align*}
\frac{p : P}{dT \, p : \text{decidable } P \, \text{true}} & \quad \frac{p : \neg P}{dF \, p : \text{decidable } P \, \text{false}}
\end{align*}
\]

Thus \( \text{decidable } P \, b \) denotes the fact that provability of \( P \) is decidable by the algorithm \( b \). As a side note, the above inductive type can be extended to account for partial algorithms, for example, three-valued ones, where one of the values stands for the undefined result.

Common decision procedures may be based on various notions of derivative. For example, decision procedures for regular expression subtyping may be based on deterministic [10] or non-deterministic derivatives [11]. Both kinds of derivative can be implemented in the type theory of Coq [12, 13].

6 Monadic substitution

We implemented in Coq a generic notion of symbolic substitution introduced in [14] for untyped lambda terms. It is based on the notion of universe of types, that is, a function space \( A \rightarrow \star \) where \( A \) can be any given type and \( \star \) is the polymorphic type of all types. \( A \) is said to index the type \( \star \). For effective indexing, the index type should be countable, and for that, it suffices to consider the type \( \mathbb{N} \) of natural numbers. Following McBride [15], we call the resulting type of universe \( \text{stuff} : \star \):

\[
\text{stuff} : \star
\]

\[
\text{stuff} = \mathbb{N} \rightarrow \star
\]

For a given \( n \), the intended meaning of \( \text{stuff } n \) is \( \text{stuff with } n \text{ variables} \).

In the foundation of the method, there is a type of monadic structure called \( \text{kit} \) [15]:

\[
\text{kit} : \text{stuff } \rightarrow \text{stuff } \rightarrow \star
\]

\[
\begin{align*}
\text{var} : \forall n. \mathcal{I}_n \rightarrow U \, n & \quad \text{lift} : \forall n. \, U \, n \rightarrow T \, n & \quad \text{wk} : \forall n. \, U \, n \rightarrow U \, (1 + n)
\end{align*}
\]

\[
\text{Kit } \text{var} \, \text{lift} \, \text{wk} : \text{kit } U \, T
\]

A substitution of type \( \text{sub } T \, m \, n \) is such that it applies to stuff with at most \( m \) variables and yields stuff with at most \( n \) variables. Hence a substitution is essentially an \( m \)-tuple of \( T \), that is,

\[
\text{sub} : \text{stuff } \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \star
\]
In order to establish compositionality on substitutions, we define applicative structure on substitutions which is called \texttt{subApp}:

\[
\texttt{subApp} : \text{stuff} \to \ast
\]

\[
\begin{array}{ll}
\texttt{var} : \forall n. \mathcal{I}_n \to T n \\
\texttt{app} : \forall U m n. \text{kit} U T \to T m \to \text{sub} U m n \to T n
\end{array}
\]

A straightforward substitution strategy is implemented by the function \texttt{trav} below that traverses a term \(E\) and applies a given substitution \(s\). Note that, since \(s\) is a tuple, \(s_i\) is a consistent notation for the \(i\)-th element of \(s\).

\[
\begin{align*}
\texttt{trav} & : \forall T m n. \text{kit} T \to \text{ty} \to \text{ty} m \to \text{sub} T m n \to \text{ty} n \\
\texttt{trav} K \perp s & = \perp \\
\texttt{trav} K \top s & = \top \\
\texttt{trav} K X_i s & = \text{let Kit} \_ \_ li = K \text{ in } li \_ s_i \\
\texttt{trav} K (F \to G) s & = (\text{trav} K F s) \to (\text{trav} K G s) \\
\texttt{trav} K (\mu F \to G) s & = \mu (\text{trav} K F (\text{lift_sub} K s)) \to (\text{trav} K G (\text{lift_sub} K s))
\end{align*}
\]

Here, \texttt{lift_sub} is a function that lifts a substitution to the next order, that is, shifts the indices of active variables in the substitution by +1. This function has type

\[
\forall (T U : \text{stuff}) (K : \text{kit} T U) m n. \text{sub} T m n \to \text{sub} T (1 + m) (1 + n)
\]

The traverse function allows to define an instance of the applicative structure on \text{ty} that we call \text{tyApp}, in Figure 6. In the definition of \text{tyApp}, we denoted the constructor of nameless variables by \(X\). This is a consistent notation since we defined, in Sec. 3, that \(X_i\) is of type \text{ty} \(n\) for a given natural number \(n\) and a bounded number \(i\) of type \(\mathcal{I}_n\). So, \(X\) is a function of type \(\forall n. \mathcal{I}_n \to \text{ty} n\). The monadic structure on \text{ty} can now be defined using the endomorphism constructor \texttt{stuffKit}. The function \texttt{subty}_0 substitutes the 0-th variable with a given \(\mu\)-type \(E\). It is defined using a generic substitution \texttt{sub}_0. The type of \texttt{sub}_0 is

\[
\forall (T U : \text{stuff}) (K : \text{kit} T U) n (E : T n). \text{sub} T (1 + n) \ast
\]

Weakening is specialised on \text{ty} by the function \texttt{wkty}. The function \texttt{sbst} is capture-avoiding substitution of a given term \(F\) for all occurrences of the 0-th variable of a term \(E\). At last, \texttt{unfld} unfolds a \(\mu\)-redex.

Unlike monadic presentations, named presentations of terms \textit{with holes} can be cumbersome and have limited application. Among the closest nameless but not monadic presentations are Capretta’s polynomial expressions with metavariables [8]. They require proving equality of substitutions in a context. Monadic presentations of terms allow to have substitutions as part of the construction and also allow for free to have a notion of a term with a hole. For example, it can be seen that Capretta’s tree expressions with metavariables have the same expressive power as monadic substitutions on polynomial trees.
tyApp : subApp ty
tyApp = SubApp X trav
tyKit : kit ty ty
tyKit = stuffKit tyApp
subty0 : ∀ n (E : ty n). sub ty (1 + n) n
subty0 = sub0 tyKit
wkty : ∀ n. sub ty n (1 + n) → ty n → ty (1 + n)
wkty = wkstuff tyKit tyApp
subKit : kit ty ty
subKit = Kit var (λ n. id (ty n)) (λ n. wkty (wk_sub n tyKit))
sbst : ∀ n. ty (1 + n) → ty n → ty n
sub E F = trav subKit E (subty0 F)
unfld : ∀ n. ty (1 + n) → ty (1 + n) → ty n
unfld E F = sbst (E→F) (μ E→F)

Fig. 1. The structure of substitution on ty.

7 Alternative approach to subtyping: suspension monad

A practical approach to nested induction-coinduction is presented in Agda [16]. The authors provide, at the language level, a type function ∞ : ⋆ → ⋆ which marks a given type as being coinductive. This type function has an interpretation as a suspension type constructor that can be used in functional languages with eager evaluation to model laziness. This interpretation is faithful since ∞ is supplied with delay and force operators ♭ : ∀ A. A → ∞ A and ♯ : ∀ A. ∞ A → A respectively.

One of the immediate advantages of having the suspension monad supported by the language is efficiency. This has also a positive effect on succinctness of function definitions by recursion-corecursion since the implementation includes an improved termination checker capable of inferring termination guarantees for such function definitions. This leaves behind the more syntactically oriented termination checker of Coq.

On the other hand, without support of suspension monad in Coq, we cannot follow this approach there. This is why it is very interesting to find ways to use type theory effectively without re-engineering the implementation. Also, note that currently the suspension monad does not allow to express directly type definitions which have an outer least fixed point and an inner greatest fixed point because of the way the termination checker of Agda works.

8 Conclusions

We showed how a rather simple fold encoding pattern can be used to define a prototypic subtyping relation: μ-types without products or sums. Our study is
closely related to the work of Altenkirch and Danielsson [1] who define subtyping using a suspension computation monad inspired by semantics of programming languages. The method with the suspension monad turns out to be inapplicable outside the special setting of [1]. Here, we follow a method that allows to encode infinitary subtyping by folding an inductive relation into a coinductive one, which can be done using standard type-theoretic means. It is worth noting that the presented approach of weak similarity is a natural solution to problems arising from declaring closure properties such as transitivity in coinductive relations that were discussed in [7]. Indeed, with our definitions, infinite transitivity chains do not arise.

The paper [17] discusses an issue with the current implementation of the most dependently typed systems that does not easily allow to encode bisimilarity into substitutive equality for reasoning about corecursive functions. This can be relevant to mixing induction and coinduction since mixing is essentially a fold method which, in order to work under case analysis (that is, unfolding), has to contain a reference to an abstract unfolded relation. With current implementations of dependent elimination, restoring the concrete relation behind this abstract one corresponds to a major part of work. Meanwhile, if we had elimination being able to unfold this relation automatically, this would be a clear time-saving benefit.

We can see that the traverse function \texttt{trav} defined in the paper is a prototype substitution strategy in the sense that, if we define substitution monads for other term languages and subtype relations of interest, the traverse function may carry some non-trivial operational meaning such as that of various matching strategies for (possibly extended) regular expressions. One of such interesting languages is the language of regular types [15], that is, recursive types with product and sum datatype constructors, which can be viewed as generalising regular expressions with non-terminating left-recursion.

Acknowledgements. I would like to thank Keiko Nakata and Niels Anders Danielsson for their help and advice regarding theorem proving and provers. The research is supported by the research fellowship EU FP7 Marie Curie IEF 253162 ‘SImPL’.

References


61
Static Detection of Deadlocks in Erlang

Maria Christakis\(^1\) and Konstantinos Sagonas\(^{1,2}\)

\(^1\) School of Electrical and Computer Engineering, National Technical University of Athens, Greece
\(^2\) Department of Information Technology, Uppsala University, Sweden
{mchrista,kostis}@softlab.ntua.gr

Abstract. We address the problem of detecting possible kinds of deadlocks in Erlang programs using static analysis. Our analysis is completely automatic, fast and sound for defect detection: it is effective in detecting deadlocks while avoiding false alarms. We have integrated our analysis in dialyzer, a widely used tool for detecting software defects in Erlang programs, and demonstrate its effectiveness on open-source libraries and applications of considerable size. Despite the fact that most of these applications have been used over a long period of time and are reasonably well-tested, our analysis has detected a number of previously unknown deadlocks in their code that may have devastating effects at runtime.

1 Introduction

A deadlock is an unintended condition under which a number of actions are stuck on some synchronization primitive waiting for each other to proceed. Deadlocks can have detrimental effects on a system. Concurrent programs that are particularly prone to deadlocks usually employ a wide range of synchronization primitives to control thread interactions and avoid other concurrency errors, such as data races. On top of that, the natural shift to multi-core machines, which have nowadays become ubiquitous, makes many more thread interactions possible, thus making the occurrence of deadlocks more likely.

Erlang [1] is a functional programming language that has been designed with the aim of simplifying concurrent programming. The language avoids the complicated management of threads and the error-prone use of explicit locking operations on shared variables, since its concurrency model is based on light-weight, user-level processes that communicate using asynchronous message passing. However, Erlang does not avoid all problems associated with concurrent execution. In particular, it provides for interactions and communication patterns between processes that may be stuck in their execution until some conditions are met, hence allowing certain kinds of deadlocks in programs. In addition, as Erlang’s primary application area is in large-scale, reactive control systems that create an unbounded number of processes, such as server applications, these process interactions become complex and extremely hard to predict. Not only that, but also deadlocks may remain unexposed during testing and when revealed, it is quite distressing to reproduce them, let alone find their cause.
To ameliorate this situation, we have identified possible kinds of deadlocks in concurrent Erlang programs, and have designed an effective analysis that detects them and brings them to the attention of the programmers. Besides tailoring the analysis to the characteristics of the language, the main challenges for our work have been to develop an analysis that: 1) is completely automatic and requires no guidance from its user; 2) strikes a proper balance between soundness and completeness in order to be: 3) fast and scalable. As we will soon see, we have achieved these goals.

The contributions of our work are as follows:

- we document the most important kinds of deadlocks in Erlang programs;
- we present an effective and scalable analysis that detects these errors, and
- we demonstrate the effectiveness of our analysis by running it on a set of widely used and reasonably well-tested libraries and open source applications and reporting a number of previously unknown deadlocks in their code bases.

The next section overviews the Erlang language and the defect detection tool which is the implementation platform for our work. Sect. 3 describes possible kinds of deadlocks in Erlang programs, followed by Sect. 4 which presents in detail the analysis we use to detect them. The effectiveness of our analysis is evaluated in Sect. 5 and the paper ends with a review of related work (Sect. 6) and some final remarks.

2 Erlang and Dialyzer

Erlang [1] is a strict, dynamically typed functional programming language with support for concurrency, communication, distribution, fault-tolerance, on-the-fly code reloading, automatic memory management and support for multiple platforms. Erlang’s primary application area has been in large-scale embedded control systems developed by the telecom industry, but its uses have expanded to application areas such as web services, online commerce, gaming, banking, etc. The main implementation of the language, the Erlang/OTP (Open Telecom Platform) system from Ericsson, has been used quite successfully both by Ericsson and by other companies around the world to develop software for large commercial applications. Nowadays, applications written in the language are significant, both in number and in code size, making Erlang one of the most industrially relevant declarative languages.

Erlang’s main strength is that it has been built from the ground up to support concurrency. Its concurrency model differs from most other programming languages out there as it is not based on shared memory but on asynchronous message passing between extremely light-weight processes (lighter than OS threads). Erlang comes with a spawn family of primitives to create new processes, and with ! (send) and receive primitives for interprocess communication via message passing. Any data can be sent as a message and processes may be located on any machine. Each process has a mailbox, essentially a message queue, where each message sent to the process will arrive. Message selection from the mailbox
occurs through pattern matching. To support robust systems, a process can register to receive a message if another one terminates. Erlang provides mechanisms for allowing a process to timeout while waiting for messages, a try/catch-style exception mechanism for error handling, and ways to organize processes in supervision hierarchies to restart or take over the duties of dead or unresponsive processes.

Since 2007, the Erlang/OTP distribution includes a static analysis tool, called dialyzer [2,3], for finding software defects (such as type errors, exception-raising code, code which has become unreachable due to some logical error, etc.) in single Erlang modules or entire applications. Nowadays, dialyzer is used extensively in the Erlang programming community and is often integrated in the build environment of many applications. The tool is totally automatic, easy to use and supports various modes of operation: command-line vs. GUI, starting the analysis from source vs. byte code, focussing on some kind of defects only, etc. In sequential programs notable characteristics of dialyzer’s core analysis are that it is sound for defect detection (i.e., it produces no false alarms), fast and scalable. Its core analyses that detect defects are supported by various components for creating and manipulating function call graphs for a higher-order language, control-flow analyses, efficient representations of sets of values, data structures optimized for computing fixpoints, etc. Since November 2009, dialyzer’s analysis has been enhanced with a component that automatically detects data races in Erlang programs [4]. Most recently, we have presented a static analysis that is able to detect some commonly occurring kinds of message passing errors in languages with dynamic process creation and communication based on asynchronous message passing [5]. This analysis is integrated in the development version of dialyzer. Before we describe how we extended dialyzer’s analyses to detect another class of concurrency errors, namely deadlocks, let us first see the kinds of deadlocks that may exist in Erlang programs.

3 Deadlocks in Erlang

In this paper, we consider a deadlock to be the condition under which the progress of a number of processes is prevented due to some dependence either on each other or on other processes. In this sense, we identify two kinds of deadlocks, namely the communication and the behaviour deadlocks. We present them in the following two subsections.

3.1 Communication Deadlocks

Let us examine Erlang’s main concurrency primitives in more detail:

Spawn The spawn primitive creates a process and returns a process identifier (pid) for addressing the newly spawned process. The new process executes the code of the function denoted in the argument of the spawn. In the example program shown in Fig. 1, two processes are spawned that will execute
-module(ping_pong).
-export([play/0]).

play() ->
    Ping = spawn(fun ping/0),
    spawn(fun() -> pong(Ping) end).

ping() ->
    receive
        pong -> ok
    end.

pong(Ping) ->
    Ping ! pong,
    receive
        ping -> ok
    end.

Fig. 1. Example program with communication deadlock

the code of functions ping/0 and pong/1. We will refer to these processes as the ping and pong processes respectively.

Send The expression Pid ! Msg sends the message Msg, that may refer to any valid Erlang term, to the process with pid Pid in a non-blocking operation. In our example program, the pong process sends the message pong to the ping process.

Receive Messages are received with the receive construct. Each process has its own mailbox for messages it receives. A mailbox functions as a queue in the sense that any new messages are placed at the end of the mailbox. When a process executes a receive, the first message in the mailbox is matched against the patterns of the receive in sequential order. If the message matches some pattern, it is removed from the mailbox and the actions corresponding to the matching pattern are executed. However, if the message does not match, it is kept in the mailbox and the next message is tried instead. If this matches any pattern, it is removed from the mailbox while keeping the previous and any other messages in the mailbox. In case the end of the mailbox is reached and no messages have been matched, the process blocks (i.e., stops execution) and waits to be rescheduled to repeat this procedure. In the example program of Fig. 1, the ping process will receive a pong message from the pong process but the latter will block as its mailbox will be empty.

Misuse of these concurrency and communication primitives may lead to communication deadlocks. In such deadlocks, messages are the resources for which processes wait. We therefore define a communication deadlock as the condition under which one or more processes block on some receive statement. More specifically, a communication deadlock occurs in the following cases:

No messages A receive statement in the code executed by some process blocks because the process mailbox is empty. This defect could reveal a
set of processes mutually waiting for messages from each other without any process in the set ever sending a message.

**Messages of the wrong kind** A `receive` statement in the code of some process blocks because the process mailbox contains messages of different kinds than the ones expected by the `receive`. Currently, such a defect, apart from blocking the process execution, can have devastating effects on a running system, overflowing the mailbox of some process and bringing the node down.

Such types of concurrency defects might have disastrous consequences in any system, let alone the safety-critical systems developed in the telecommunications sector. But these are not the only kind of deadlocks possible in Erlang programs.

### 3.2 Behaviour Deadlocks

Erlang/OTP comes with some commonly employed concurrency design patterns, called *behaviours*. For example, the client-server model, comprising a central server and an arbitrary number of clients, is frequently used for resource management, i.e., the clients share a common resource managed by the server. Usually, the clients and servers in each instance of the client-server model share similar structure patterns, and behaviours are formalizations of these patterns. The standard Erlang/OTP behaviours include the implementation of servers in client-server relations, finite state machines, event handlers and supervisors in supervision trees. User-defined behaviours may also be implemented.

When a behaviour is used in the implementation of a process, the code is divided into a generic and a specific part, called the *behaviour* and *callback modules* respectively. For built-in behaviours, the behaviour module is part of Erlang/OTP while the callback module is implemented by the user. For instance, for the creation of a server process, the user must write a callback module that exports a predefined set of callback functions.

In the example program of Fig. 2, `gen_server` and `server` are the behaviour and callback modules respectively. The latter provides an interface to the server for manipulating a counter and a set of callback functions. The server interface includes functions `make_counter/1`, `count_down/1` and `set/2` that, in the order mentioned, create a counter initialized to zero, perform a countdown to zero and set the counter to a given value. The callback functions are implicitly called by the following functions of the behaviour module:

- `gen_server:start_link(ServerName, Module, Args, Options)` which creates a generic server process that calls function `Module:init/1` with arguments `Args` to initialize. The server is registered, either locally or globally, under a name specified in `ServerName`.
- `gen_server:call(ServerRef, Request)` which makes a synchronous call to the server with reference `ServerRef` by sending a request and waiting until a reply arrives or a timeout occurs. The default value for the timeout is 5000 ms. If no reply is received within this specified time, the call fails. The server calls function `Module:handle_call/3` to handle the request. Here, the reference `ServerRef` is determined by the registered name `ServerName` passed
-module(server).
-behaviour(gen_server).
-export([make_counter/1, count_down/1, set/2]).
-export([init/1, handle_call/3, handle_cast/2]).

make_counter(?T) ->

count_down(?T) ->

set(?T, N) ->
    gen_server:cast(?S, {?T, N}).

init(?T) ->
    ets:new(?T, [named_table, public]),
    ets:insert(?T, {counter, 0}),
    {ok, feeling_good}.

handle_call(?T, _From, St) ->
    [{counter, N}] = ets:lookup(?T, counter),
    case N of
        0 ->
            ok;
        _ ->
            ets:insert(?T, {counter, N - 1}),
            gen_server:call(?S, ?T, infinity)
    end,
    {reply, 0, St}.

handle_cast(?T, N, St) ->
    ets:insert(?T, {counter, N}),
    {noreply, St}.

Fig. 2. Example program with behaviour deadlock

to gen_server:start_link/4 and Module refers to the callback module also
passed to the same behaviour function.

gen_server:call(ServerRef, Request, Timeout) which has the same function-
ality as gen_server:call/2. The only difference is that this call allows the
user to specify how long to wait for a reply. Timeout may be either a positive
integer specifying the number of milliseconds to wait or the atom infinity
meaning that the call will wait indefinitely for a reply.

gen_server:cast(ServerRef, Request) which makes an asynchronous call to
the server with reference ServerRef by sending a request and returning
immediately. The server calls function Module:handle_cast/2 to handle the
request.

The astute reader has already noticed that the count_down/1 function of
Fig. 2 will fail due to a deadlock: It calls function gen_server:call/2 and
waits for a reply. However, the handle_call/3 function, that handles this re-
quest, instead of calling itself recursively in order to successively reduce the
value of the counter, sends another synchronous request to the server with the
gen_server:call/3 call. If the example were not made up, we would assume
that the programmer considered the gen_server:call/3 and handle_call/3
functions equivalent, forgetting that the former is the synchronous version of
the latter. As a result, the gen_server:call/2 function becomes synchronously
recursive, deadlocks and fails when the default timeout occurs. Taking the ex-
ample program a step further, it is possible for more than one servers to be
involved in such a deadlock in case a server’s synchronous request issues addi-
tional such requests to other servers that analogously issue synchronous requests
to the server that initially triggered them. We therefore define a behaviour dead-
lock as the condition under which two or more synchronous calls are mutually
waiting for each other. Usually, when such a deadlock occurs, a timeout comes
to the rescue.
Behaviour deadlocks may be caused by any synchronous behaviour functions of Erlang/OTP since, apart from being synchronous, they require the interference of the user for the implementation of the callback functions and are thus prone to errors. Having presented these two kinds of deadlocks in Erlang, which are also the categories of deadlocks that our analysis detects, let us now present the details of the analysis.

4 The Analysis

Statically detecting the deadlocks we described in the previous section is not trivial. In order to detect communication deadlocks in a higher-order language with unlimited process creation and asynchronous message passing such as Erlang, the communication topology of processes needs to be determined in a fairly precise way. On the other hand, the behaviour deadlock detection requires a quite different approach. Concrete information about the Erlang behaviours and their functionality must be provided to the analysis, which will use this information for the creation of a wait-for graph, the basis for detecting these errors. We have designed and implemented such analyses and describe them in this section. Although we describe them as being distinct, the actual implementation blurs the lines of this distinction for efficiency reasons.

We have integrated our analyses in dialyzer because many of the components that it relies upon were already available or could be easily extended to provide the information that the analyses need. The analyses start with the user specifying a set of directories/files to be analyzed. Rather than operating directly on Erlang source, all of dialyzer’s passes operate at the level of Core Erlang [6], the language used internally by the Erlang compiler. Core Erlang significantly eases the analysis of Erlang programs by removing syntactic sugar and by introducing a let construct which makes the binding occurrence and scope of all variables explicit.

4.1 Detection of Communication Deadlocks

Conceptually, the analysis for the detection of communication deadlocks has three distinct phases: an initial phase that scans the code to collect information needed by the subsequent phases, a phase where a communication graph is constructed, and a phase where communication deadlocks are detected.

In the first phase of the analysis, as the source code is translated to Core Erlang, dialyzer constructs the control-flow graph (CFG) of each function and function closure that will later be traversed in search of concurrency primitives. Dialyzer then uses the escape analysis of Carlsson et al. [7] to determine values, in particular closures, that escape their defining function. Given this information, dialyzer also constructs the inter-modular call graph of all functions and closures, so that subsequent analyses can use this information to speed up their fixpoint computations. Based on both the escape analysis and the call graph, the analysis identifies the processes that might be created at runtime. Besides
Fig. 3. Communication graph of example program with communication deadlock control-flow, the analysis also needs data-flow information and more specifically it needs information on whether variables can possibly refer to the same process identifier or not. This information is computed and explicitly maintained by the sharing/alias analysis component in dialyzer’s race analysis [4]. In addition, our analysis exploits the fact that dialyzer computes type information at a very fine-grained level [8] to decide whether messages match their receiving patterns.

The second phase of the analysis determines the interprocess communication topology in the form of a graph. Each vertex of the graph represents an escaping function whose code may be run by a separate process at runtime. This information is computed by a pre-processing step during the construction of the call and control-flow graphs. The code of any function that is either a root node in the call graph or an argument to a spawn is assumed to be executed by a separate process. For our example program, the communication graph will contain three nodes, for functions play/0 and ping/0 and for the closure. Every edge of the communication graph is directed and corresponds to a communication channel between two processes. Naturally, its direction of communication is from the source to the target process, meaning that messages are sent in that direction. Each edge is annotated with the type information of the messages that are sent through the channel. Naturally, its direction of communication is from the source to the target process, meaning that messages are sent in that direction. Each edge is annotated with the type information of the messages that are sent through the channel. In order to determine the graph edges, we need to inspect every possible execution path of the program for messages that are passed between processes. To this end, the analysis traverses the CFGs of the functions corresponding to the vertices in the communication graph using depth-first search. If the traversal finds a send operation to some pid, the analysis takes variable sharing into account to determine the recipient process that this pid refers to, thus identifying the target vertices of each edge. In the end, this traversal creates the complete set of edges in the communication graph. For the code of Fig. 1, the communication graph has one edge from the closure to vertex ping/0 annotated with pong since a pong message is sent from the process executing the code of the closure to the ping process. The communication graph for this example program is illustrated in Fig. 3.

At the final stage of the analysis, the CFG of each function that corresponds to a vertex in the communication graph is traversed anew to detect any communication deadlocks. Each vertex in the communication graph has an in-degree that is either equal to or greater than zero. A vertex with in-degree equal to zero indicates that no messages are sent to the process it represents. Hence, the traversal of the CFG emits a warning for each receive construct it encounters. A vertex with in-degree greater than zero indicates that messages are sent to
the process and the analysis determines whether these messages will be received. In case the process expects to receive messages (i.e., there are receives in the CFG), the analysis takes into account the type information of the messages and the receive patterns in order to decide whether they match. In short, at the end of the CFG traversal, warnings are emitted for receive constructs that do not have matching patterns for any messages. For the example program, the analysis inspects the CFG of the closure, which has in-degree zero, and finds that there is a receive in the code executed by the process. Consequently, it emits a warning with the filename and line number of the receive reporting a communication deadlock.

This part of the analysis is similar to the analysis for the detection of message passing errors [5], as we consider any message passing errors that involve a blocking receive to be communication deadlocks. The interested reader may refer to that publication for a more detailed description of the analysis. The optimization ideas and the techniques to avoid false alarms in case the available static information is too limited to construct the exact interprocess communication graph presented there, are also employed by the communication deadlock analysis in order to speed up its performance and make it sound for defect detection.

4.2 Detection of Behaviour Deadlocks

The deadlock detection analysis also has three phases: an initial phase that scans the code and collects information, a phase where a wait-for graph is constructed, and a phase where behaviour deadlocks are detected.

The first phase of the analysis collects information that will be used for the construction of the wait-for graph in the next phase, such as dialyzer's type information, the CFGs of each function and function closure and the intermodular call graph. Besides this, during the source code translation to Core Erlang, information is obtained on which behaviours, if any, are implemented by the user. For the example of Fig. 2, the analysis finds that the generic server (gen_server) design pattern is implemented since it is declared in the behaviour attribute of the module. In addition, calls to any behaviour functions need to be identified. To compute this information, the analysis first refers to its hard-coded set of calls for each implemented behaviour. Then, during the construction of the function CFGs, it collects a set of program points containing any of these calls. The behaviour set, i.e., the set of behaviour calls, for the example program contains the gen_server:start_link/4, gen_server:call/2,3 and gen_server:cast/2 calls. Finally, the analysis refers once more to its hard-coded information to generate a subset of the behaviour set, containing only the synchronous calls. The gen_server behaviour offers two families of synchronous calls, the gen_server:call/2,3 and gen_server:multi_call/2,3,4 calls, that are all handled by the same callback function, the handle_call/3 function. Based on this information, the analysis filters the behaviour set to create the synchronous subset containing the gen_server:call/2,3 calls.

The second phase of the analysis constructs a wait-for graph, which is essentially a call graph of synchronous calls. Every vertex of the graph represents a
callback function that handles the requests of a number of synchronous calls. As we have already mentioned, both of the calls in the synchronous set are handled by the same callback function. Thus the wait-for graph for our example program has only one vertex annotated with the callback function and the program points containing the synchronous calls that it handles. However, the analysis does not have any information about the callback module, i.e., the module where the `handle_call/3` function is defined. To identify this module, the analysis first uses its hard-coded information to establish which argument of the synchronous calls refers to the server and then exploits dialyzer’s type information to extract the server name. In Fig. 2, the `gen_server:call/2,3` calls refer to the server `server`, an atom defined in a macro definition. Now, the analysis uses the behaviour set to look for any calls that might register the server under this name. In this case, `gen_server:start_link/4` is the only call that could register the server, as described in Sect. 3.2, and the type information of its first argument confirms the name. From the type information of its second argument, it is established that the `server` module is the callback module for this behaviour. Thus the analysis infers that the requests of the `gen_server:call/2,3` calls are handled by the `server:handle_call/3` function and the vertex of the wait-for graph is created. Note that if the type information for the behaviour calls were not precise enough to correctly identify the callback module, the analysis would not proceed to avoid emitting any false alarms. In the wait-for graph, there is a directed edge from vertex $V_1$ to vertex $V_2$ if there exists a synchronous call whose request is handled by the callback function of $V_1$ that must wait for a synchronous call handled by the callback function of $V_2$ to return. The edges are determined by checking whether there is a path in the inter-modular call graph from the callback function in each vertex of the wait-for graph to any caller function of a synchronous call in the same or any other vertex of the graph. If such a path exists, an edge is added from the vertex of the callback function to the vertex of the synchronous call. The path for the example program is trivial since the callback function `handle_call/3` is also the caller function of `gen_server:call/1/3`. The wait-for graph for this program is shown in Fig 4.

The final phase of the analysis uses the wait-for graph in order to detect behaviour deadlocks. A program might deadlock if and only if there is a directed cycle in its wait-for graph. Consequently, the analysis searches the graph for the existence of cycles. If such cycles exist, it reports the synchronous calls that are mutually waiting for each other. As expected, the wait-for graph of Fig. 4 has a cyclic wait that involves the `gen_server:call/2,3` calls.
A dependency graph for the behaviour deadlock analysis is shown in Fig. 5. The phases of the analysis are indicated with different colors. Let us now evaluate the effectiveness of these techniques on a suite of large, widely used Erlang applications.

5 Experimental Evaluation

The analysis we described in the previous section has been fully implemented and incorporated in the development version of dialyzer. We have paid special attention to integrate it smoothly with the existing analyses, reuse as much of the underlying infrastructure as possible, and fine-tune the analysis so that it incurs relatively little additional overhead to dialyzer’s default mode of use. The core of the deadlock analysis is about 2,500 lines of Erlang code and the user can turn it on either via a GUI button or a command-line option.

We have measured the effectiveness and performance of the analysis by applying it on a corpus of Erlang code bases of significant size; in total more than a million lines of code. As these code bases have been developed and tested over a long period of time, it is perhaps not surprising that our analysis did not find deadlocks in most of them. In fact, many Erlang developers admit to having run into and corrected such errors. An indicative example is that of Reia\(^4\), a hybrid object/actor language for the Erlang VM: at some point during its implementation, there was a behaviour deadlock involving two generic servers synchronously calling each other that was later eliminated. Still, there are Erlang/OTP libraries and applications for which the analysis has detected possible deadlocks currently

\(^3\) The source of Erlang/OTP distribution alone is about 800k lines of code.
present in their code. A short description of these code bases appears in Table 1; most are heavily used and reasonably well-tested. For open source applications, we used the code from their public repositories at the end of March 2011.

**Table 1. Applications for which the analysis detected deadlocks**

<table>
<thead>
<tr>
<th>Application libraries from the Erlang/OTP R14B02 distribution</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>snmp</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Open source Erlang applications</td>
<td></td>
</tr>
<tr>
<td>dynomite</td>
<td>A Dynamo clone</td>
</tr>
<tr>
<td>effigy</td>
<td>A mocking library for testing</td>
</tr>
<tr>
<td>log.roller</td>
<td>A distributed logging system</td>
</tr>
<tr>
<td>yatsy</td>
<td>Yet Another Test Server — Yaws compatible</td>
</tr>
<tr>
<td>zotonic</td>
<td>A content management system</td>
</tr>
</tbody>
</table>

**Table 2. Effectiveness of the deadlock analysis**

<table>
<thead>
<tr>
<th>Application</th>
<th>LOC</th>
<th>CD</th>
<th>BD</th>
</tr>
</thead>
<tbody>
<tr>
<td>snmp</td>
<td>56,728</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>dynomite</td>
<td>19,381</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>effigy</td>
<td>1,288</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>log.roller</td>
<td>2,539</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>yatsy</td>
<td>2,356</td>
<td>-</td>
<td>23</td>
</tr>
<tr>
<td>zotonic</td>
<td>68,462</td>
<td>-</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2 shows the lines of code (LOC) for each application and the number of blocking program points identified by the analysis. These are shown categorized as in Sect. 3: namely, as related to a communication deadlock (CD) because they involve a blocking receive, or a behaviour deadlock (BD) because there is a synchronous call that will either timeout or wait forever. As can be seen in the table, the analysis detects a number of errors, which may be detrimental to the functionality and robustness of these applications. We have manually examined the source code of these applications and all these problems are genuine bugs under certain runtime conditions that are dependent on the program input and the chosen execution paths. More details about these errors may be found on dialyzer's website: [http://www.softlab.ntua.gr/dialyzer/](http://www.softlab.ntua.gr/dialyzer/).

Regarding performance, we deliberately did not include measurements of the additional time and memory overhead of the deadlock detection component of the analysis as it is too small to care about. Given that the analysis is totally
automatic and smoothly integrated in a defect detection tool which is widely used by the community, we see very little reason not to use it regularly when developing Erlang programs.

6 Related Work

The problem of detecting deadlocks in concurrent programs is fundamental and well studied. In the literature one can find various approaches either for shared-memory, distributed [9] or database systems [10]. Work on deadlock detection for the former systems includes many static approaches. Boyapati et al. [11] have presented a type based approach that allows programmers to specify a partial order among locks and guarantees that well-typed programs are free of data races and deadlocks. A number of data-flow analyses have also been proposed. Among them, approaches by von Praun [12], Williams et al. [13] and Engler and Ashcraft [14] rely on the computation of a static lock-order graph and report cycles in the graph as possible deadlocks. Similarly, our analysis for behaviour deadlock detection constructs a static graph and searches it for cycles. In distributed and database systems, most approaches are dynamic and involve cycle detection in wait-for graphs. In these approaches, the main points of interest are the efficiency of the cycle detection algorithms and the methods employed for the construction and maintenance of the wait-for graph.

Regarding communication, some researchers have proposed using effect-based type systems to analyze the communication behaviour of message passing programs; an early such work is the analysis by Nielson and Nielson for detecting when programs written in CML have finite topology [15]. There has also been a number of abstract interpretation based analyses that are closer in spirit to the analysis we employ for the detection of communication deadlocks. Mercourff designed and implemented an analysis for CSP programs with a static structure based on an approximation of the number of messages sent between processes [16] and Martel and Gengler an analysis that statically determines an approximation of the communication topology of a CML program [17]. The abstract interpretation based whole program analysis of Colby uses control paths to identify threads [18]. Unlike earlier work which collapsed multiple threads created at the same spawn point to a single approximate thread, control paths are able to distinguish multiple threads created at the same spawn point and thus compute a more precise interprocess communication topology of a program. Still, the precision problem was not completely solved. A more precise, but also more complex and less scalable, control-flow analysis was proposed by Martel and Gengler [17]. In their work the accuracy of the analysis is enhanced by building finite automata that eliminate some impossible communication channels and aid in computing the possibly matching emissions for each reception, and thus the possibly received values.

In 2009, Claessen et al. proposed a method to detect race conditions in Erlang programs by employing property-based testing using QuickCheck and a special purpose randomizing user-level scheduler for Erlang called PULSE [19]. The
PULSE scheduler controls its processes by randomly picking only one of them to run at a time. As an additional benefit, this design allows PULSE to detect communication deadlocks when all of its randomly interleaved processes are blocked waiting on some receive and no messages are being sent to any of the blocked processes. To the best of our knowledge, this has been the only attempt to detect deadlocks in Erlang programs so far. While we prefer our method because it is more scalable and analyzes the entire program instead of random process schedules, QuickCheck and PULSE may find deadlocks that our tool would suppress for fear of emitting false alarms in case the available static information were not precise enough.

7 Concluding Remarks

We have showed kinds of deadlocks that Erlang programs can exhibit and have presented a static analysis that detects them. Our analysis is fast, robust and uses effective techniques to achieve a proper balance between precision and performance. By implementing this analysis in a publicly available and commonly used tool for detecting software defects in Erlang programs, we were able to detect a number of previously unknown deadlocks in widely used and reasonably well-tested applications written in Erlang, as shown in the experimental evaluation section of the paper. By identifying possible kinds of deadlocks in Erlang programs, we also contribute in a concrete way to raising the awareness of the Erlang programming community on these errors.

In the future, we hope that Erlang developers will be watching out for these errors when programming. We also plan for our analysis to be included in an upcoming release of Erlang/OTP, thus acquiring its place in the developer’s tool suite.

References

Size Analysis of Higher-Order Functions

Attila Gobi ∗1, Olha Shkaravska2, and Marko van Eekelen∗∗2,3

1 Faculty of Informatics, Eötvös Loránd University Budapest
2 Institute for Computing and Information Sciences, Radboud University Nijmegen
3 Open University of the Netherlands, Heerlen

Abstract. We present a lambda-calculus that formalizes the relations between the sizes of arguments and the sizes of the corresponding results of functions in a higher-order polymorphic strict functional language. On top of usual constructions we consider two operators for finite maps: List, that defines a (higher-order) finite maps, and Shift. Intuitively, size expressions are abstract interpretations of programs in the natural arithmetic.

To prove normalization and diamond (modulo integer axiomatics) property of the calculus we show that it can be expressed in System F.

1 Introduction

We present a calculus that formalizes the relations between the sizes of arguments and the sizes of the corresponding results of functions in a higher-order polymorphic strict functional language. Informally, the calculus extends the lambda-calculus with arithmetic operations and two operators for finite maps: List, that defines a (higher-order) finite maps, and Shift. To our knowledge, the novelty of our approach is in using finite maps and the two operators above to present size of lists. In the future we will consider possibility to infer polynomial size dependencies for higher-order shapely functions, using polynomial interpolation.

Verification conditions we obtain as the result of syntax-directed stage of type-checking, are (conditional) equations in the combination of three theories: lambda-calculus, integer ring and finite maps. From the theory of finite maps we need the extensionally axiom that looks like

\[ f = g \iff \text{Dom}(f) = \text{Dom}(g) \land \forall n \in \text{Dom}(f). f(n) = g(n) \]

and the rewriting definition of Shift- and List-operators.

This research continues the series of work on size analysis of first-order strict functional languages where annotation inference is based on polynomial interpolation[7]. At the end of this paper we show that similar test-and-interpolate heuristic, which hints possible polynomial dependencies between sizes, is applicable for higher-order functions as well.

∗ The research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003)

∗∗ The second and the third authors are supported by Artemis Joint Undertaking in the CHARTER project, grant-nr. 100039
2 Size calculus

Syntax of expressions in the calculus is given by the grammar on figure 1. We call them size expressions.

Now we consider a few simple examples to give an idea behind our formalization. We begin with an integer literal, eg. 42. We assume that it does not have size, so we assign the expression \textit{Unsized} to it.

The size of a list is expressed by the combinator \texttt{List}. For instance, the size of the list \([1]\) is given by \texttt{List 1 (\(\lambda x.\,\text{Unsized}\))}. Here the first argument of \texttt{List} denotes the length of the list while the second is a lambda abstraction expressing the sizes of the elements of the list. \(\lambda\)-bound variable \(x\) corresponds to the position of the element in a list. For instance, in the expression \texttt{List \(n\) \((\lambda x.\,e(x))\)} \(e(n - 1)\) represent the size of the head, \(e(n - 2)\) represents the size of the element next to the head, \(e(0)\) represents the length of the tailing element. \(^4\) \(e(x)\) is a finite map defined on \(0, \ldots, n - 1\).

To define higher-order size expressions for functions we use two sorts of \(\lambda\)-abstraction. For type list we use \(\Lambda\) abstraction, for other types we use \(\lambda\).

For example, the size expression for a function of type \texttt{Int \to Int} is \(\lambda x.\,\text{Unsized}\), since the size of its result is unsized regardless of its argument. Another simple example is the identity function of type \(\alpha \to \alpha\). Its size expression is \(\lambda x.\,x\), which expresses the fact it does not change the size of its argument.

For list arguments we use \(\Lambda\). In a way, it is the “inverse” of \texttt{List}, as \texttt{List} can be seen as a data constructor of a pair, while \(\Lambda\) is the corresponding pattern matching. In this paper it is defined by the following reduction:

\[
(\hat{\lambda}_y^x.e)(\texttt{List } a\ b) \to (e[x := a, y := b])
\]

For an example of applying this definition see the comment after the match-rule in Section 3.

In this way we can present the size expression for the result of a function by giving a size expressions of its arguments. In the following example the function \texttt{addone} takes its argument \(l' : L(\texttt{Int})\) and appends 1 to the list:

\[\texttt{addone } l' = \texttt{cons } 1\ l'\]

The simplest way to express its size dependency is:

\[
\hat{\lambda}_y^x.\,\texttt{List } (s + 1)(\lambda x.\,\text{Unsized})
\]

Now the combinator \texttt{Shift} is defined. \texttt{Shift }\(e_1\ n\ e_2\) inserts \(n\) elements of \(e_1\) before \(e_2\), so the reduction rule for the combinator is:

\[
(\texttt{Shift } e_1\ c_1\ e_2)c_2 \to \begin{cases} \frac{e_1c_2}{e_2(c_2 - c_1)} & \text{if } c_2 < c_1 \\ e_2c_2 & \text{otherwise} \end{cases}
\]

\(^4\) Note that this enumeration of list elements is “opposes” the traditional in the functional languages enumeration, where the head element has number 0, etc. The enumeration we use is more convenient in our reasoning and, for instance, simplifies significantly the match-rule.
For example, using this combinator we can define the size expression of the usual append function:

$$\lambda f.\lambda g.\text{List}(l + m)\text{(Shift} g m f)$$

### 3 Type system

Our type system does not check or infer types, but relies an underlying type system instead. For the point of view of the size checking we only need a function called Type which can give us the correct type of a function. For this reason and for the sake of readability we omit underlying types in typings at all. Thus $\tau$ in a typing $z : \tau$ is a size expression. The equality of two types $\tau_1$ and $\tau_2$ means that they have the same normal form modulo axiomatics of integer rings, if we use the reduction rules of $\lambda$-calculus and the reduction definitions of List and Shift. We will also need an equation between expression related to extensionality axiom, which we explain later.

The difference between LetFun and Let should be clarified. In the case of Let no formal parameters or recursion allowed, but it does not have explicit size signature – partial inference is used in that rule. LetFun can be recursive and can have formal parameters, but it must be annotated by a size expression and our type system can only check its type.

where

$$\text{fresh}(\alpha) = \begin{cases} 
\text{List } \tau(\text{fresh}(\beta)) & \text{if } \alpha = L(\beta) \\
\text{List } \tau(\lambda\tau.\text{'Unsized}) & \text{if } \alpha = L(\text{Int}) \\
\text{Unsized} & \text{if } \alpha = \text{Int} \\
\tau & \text{otherwise}
\end{cases}$$

where $\tau$ and $\tau'$ are fresh size variables.
We omit the cons and nil rule, because we consider them as predefined functions, so their size expressions are in the context \(\Gamma\). The size expression of nil is the following.

\[
\text{nil} : \text{List} \ 0 (\lambda x.x)
\]

Note that \(\lambda x.x\) is arbitrary and should be never evaluated in a correctly typed program. A naive version of the typing for \text{cons} is

\[
\text{cons} : \lambda x. \lambda f^s. \text{List} \ (s + 1) \ (\text{Shift} \ \ f \ s \ 0 \ x) \)
\]

where \(\{0 \mapsto x\}\) is the final map with the domain \{0\} that maps 0 to \(x\). This rule gives an insight, however it is rather semantic and we do not have syntactic tools to define finite maps explicitly. So, we use more general version of the typing:

\[
\text{cons} : \lambda x. \lambda f^s. \text{List} \ (s + 1) \ (\text{Shift} \ f s \ \lambda y. x),
\]

\[
\begin{align*}
D, (\lambda f^s, l) &\vdash \text{List} \ 0 \ \tau' ; \ \Gamma, \ l: \tau \vdash e_{\text{nil}} : \tau \\
\text{hd}, \ tl &\not\in \text{dom}(\Gamma) \quad \tau_{\text{hd}} = (\lambda f^s.f(s - 1))\tau_1 \\
\tau_{\text{tl}} &= (\lambda f^s.\text{List}(s - 1)f)\tau_1 \\
D; \ \Gamma, \ \text{hd}: \tau_{\text{hd}}, \ l: \tau_{\text{ti}} \vdash e_{\text{cons}} : \tau
\end{align*}
\]

\[
\begin{array}{c}
\text{Match} \\
D; \ \Gamma, \ l: \tau_1 \vdash \text{match} \ l \text{ with } \\
\quad \text{nil } \Rightarrow e_{\text{nil}} : \tau \\
\quad \text{cons} \ \text{hd} \ \text{tl} \Rightarrow e_{\text{cons}}
\end{array}
\]

It is worth to note that for a well-typed function the type \(\tau_1\) is reduced to the type of the form \(\text{List} \ \tau_1 \ \tau_2\) and \(\tau_1\) is reduced to the integer expression. Then the type of \(\text{hd}\), which is \((\lambda f^s.f(s - 1))\tau_1\), is reduced to \(\tau_2(\tau_1 - 1)\) according to the rewriting definition. Similarly, the type of \(\text{hd}\), which is \((\lambda f^s.\text{List}(l - 1)f)\tau_1\) is reduced to \(\text{List}(\tau_1 - 1)\tau_2\).

\[
D \vdash \tau \ \tau_1 \ldots \tau_n = \tau' \\
\rightarrow \\
D; \ \Gamma, \ f: \tau, \ x_1: \tau_1 \ldots x_n: \tau_n \vdash f(x_1 \ldots x_n): \tau' \quad \text{FUNApp}
\]

### 3.1 Examples

**append**

\[
\begin{align*}
\text{append} \ (p, q) &\vdash \lambda f^s_1. \lambda f^s_2. \text{List} \ (s_1 + s_2) \ (\text{Shift} \ f_2 \ s_2 \ f_1) = \\
&\text{match} \ p \text{ with } \\
\quad \text{nil} &\Rightarrow q \\
\quad \text{cons} \ \text{hd} \ \text{tl} &\Rightarrow \text{let} \ tl' = \text{append} \ tl \ q \\
&\text{in} \ \text{cons} \ \text{hd} \ tl'
\end{align*}
\]

Here \(p\) and \(q\) is of type \(L(\alpha)\). According to the \text{LETFUN} rule, we are creating fresh size expressions for the arguments by using the function fresh. Assuming that the fresh size expressions are \text{List} \(a\) \ and \text{List} \(c\) \ for \(p\) \ and \(q\), respectively, we need to prove the following:

\[
\begin{align*}
\text{True; append: } &\lambda f^s_1. \lambda f^s_2. \text{List} \ (s_1 + s_2) \ (\text{Shift} \ f_2 \ s_2 \ f_1), \ p: \text{List} \ a, \ q: \text{List} \ c \ \text{d} \\
&\vdash \text{match} \ldots : (\lambda f^s_1. \lambda f^s_2. \text{List} \ (s_1 + s_2) \ (\text{Shift} \ f_2 \ s_2 \ f_1)) \ (\text{List} \ a) \ (\text{List} \ c)
\end{align*}
\]

80
However it is not necessary, but in the examples we do the reductions of size expressions to make the judgements shorted and more readable:

\[ \text{True}; \ \text{append:} \lambda f_1^{s_1}, \lambda f_2^{s_2}. \text{List (} s_1 + s_2 \text{) (} \text{Shift } f_2 \ s_2 \ f_1 \text{)}, \ p: \text{List } a \ b, \ q: \text{List } c \ d \ \vdash \text{match . . . : List (} a + c \text{) (} \text{Shift } d \ c \ b \text{)} \]

The first step is proving the nil branch:

\[ a = 0; \ q: \text{List } c \ d \ \vdash q: \text{List (} a + c \text{) (} \text{Shift } d \ c \ b \text{)} \]

\[ a = 0 \vdash c = a + c \land i \geq 0 \land i < a + c \ \vdash (\text{Shift } d \ c \ b) i \]

Analyzing the cases according to the definition of \text{Shift}:

\[ a = 0 \vdash (i \geq 0) \land (i < a + c) \land (i < c) \Rightarrow d i = d i \]
\[ a = 0 \vdash (i \geq 0) \land (i < a + c) \land (i \geq c) \Rightarrow d i = b (i - c) \]
\[ a = 0 \vdash (i \geq 0) \land (i < a + c) \land (i < c) \Rightarrow \text{True} \]
\[ a = 0 \vdash (i \geq 0) \land (i < a + c) \land (i \geq c) \Rightarrow \text{False} \]

To apply the match rule, we need to calculate the size expressions for \text{hd} and \text{tl}:

\[ \tau_{\text{hd}} = b (a - 1) \quad \tau_{\text{tl}} = \text{List (} a - 1 \text{) } b \]

It is a let expression so the next step is to analyze the Let binding (which is two function applications) and infer the size of the variable \text{tl'}:

\[ \text{True} \ \vdash (\lambda x. \lambda f. \text{List (} s + 1 \text{) (} \text{Shift } f s \ \lambda y. x \text{)}) (\text{List (} a - 1 \text{) } b) (\text{List } c \ d) = \text{List (} a + c \text{) (} \text{Shift } d \ c \ b \text{)} \]

Continuing with the let body:

\[ \text{True} \ \vdash (\lambda x. x f. \text{List (} s \text{) (} \text{Shift } f s \ \lambda y. x \text{)}) (b (a - 1)) (\text{List (} a - 1 + c \text{) (} \text{Shift } d \ c \ b \text{)}) = \text{List (} a + c \text{) (} \text{Shift } d \ c \ b \text{)} \]

At the end we have to prove the equation above. After reduction we get:

\[ \text{List (} a - 1 + c + 1 \text{) (} \text{Shift (} \text{Shift } d \ c \ b \text{)} (a - 1 + c) (\lambda y. b (a - 1)) \text{) = List (} a + c \text{) (} \text{Shift } d \ c \ b \text{)} \]

It is clear that \( a - 1 + c + 1 = a + c \). For the nested part the following cases can be identified:

\[ \vdash (i \geq 0) \land (i < a + c) \land (i < a - 1 + c) \Rightarrow (\text{Shift } d \ c \ b) i = (\text{Shift } d \ c \ b) i \]
\[ \vdash (i \geq 0) \land (i < a + c) \land (i \geq a - 1 + c) \Rightarrow b (a - 1) = (\text{Shift } d \ c \ b) i \]
The first one is a tautology, while the second one can be split into two cases by applying the \textbf{Shift} rule again:

\[
\vdash (i \geq 0) \land (i < a + c) \land (i \geq a - 1 + c) \land (i < c) \Rightarrow b(a - 1) = d \ i
\]
\[
\vdash (i \geq 0) \land (i < a + c) \land (i \geq a - 1 + c) \land (i \geq c) \Rightarrow b(a - 1) = b(i - c)
\]

The first one holds because of the contradiction (eg. \(i \geq a - 1 + c\) and \(i < c\)), while the second one can be reduced to the following (which holds as well):

\[
\vdash \forall i \in \{0..a + c - 1\} : (i \geq a - 1 + c) \land (i \geq c) \Rightarrow a - 1 = i - c
\]

\textbf{map} The map function is higher-order, its first argument is a function and its second argument is a list. It maps the elements of that list with its first argument one-by-one. The interesting part of the type checking is the last step (checking \texttt{cons hd' tl'}). All others are analogous to the previous example.

\[
\text{map} (g, l) : \lambda x.\lambda s f. \text{List } s (\lambda i.x (f i)) =
\]

\[
\begin{array}{l}
\text{match } p \text{ with } \\
\quad \mid \text{nil} \Rightarrow \text{nil} \\
\quad \mid \text{cons } hd \ tl \Rightarrow \text{let } tl' = \text{map } g \ tl \\
\quad \quad \text{in } \text{let } hd' = g \ hd \\
\quad \quad \text{in } \text{cons } hd' \ tl'
\end{array}
\]

Let’s assume that the fresh size variables are \(a\) and \(\text{List } bc\), than the type environment before the last step is:

\[
\Gamma = \{ g : a, \ l : \text{List } bc, \ hd' : a(c(b - 1)), \ tl' : \text{List } (b - 1) (\lambda i.x(c i)), \ldots \}
\]

So we have to show the following:

\[
\vdash (\lambda x.\lambda f s. \text{List } (s + 1) (\text{Shift } f s (\lambda y.x))) \ (a(c(b - 1))) \ (\text{List } (b - 1) (\lambda i.(a(c i)))) =
\]

\[
\text{List } b (\lambda i.a(c i))
\]

\[
\vdash \text{List } (b - 1 + 1) (\text{Shift } (\lambda i.(a(c i))) (b - 1) (\lambda y.a(c(b - 1)))) =
\]

\[
\text{List } b (\lambda i.a(c i))
\]

The non-trivial part here is:

\[
\vdash (j \geq 0) \land (j < b) \land j \geq b - 1 \Rightarrow (\lambda y.a(c(b - 1)))(j - b + 1) = (\lambda i.a(c i))j
\]
\[
\vdash (j \geq 0) \land (j < b) \land j \geq b - 1 \Rightarrow a(c(b - 1))) = a(c j)
\]
\[
\vdash (j \geq 0) \land (j < b) \land j \geq b - 1 \Rightarrow b - 1 = j
\]

\textbf{t3} The most interesting question is how can the size expressions handle such a polymorphism when an argument can be a list and even a function. To demonstrate this case we define the following function:

\[
t3 (g, x) : \lambda f.\lambda x.f(f(fx)) = g(g(g x))
\]
It’s easy to check the type of this function so it is left for the reader. The interesting part is when we use this function in different kinds of expressions:

```plaintext
let t = t3 t3 in t addone
```

In this example the inferred type for `t` is

\[
\lambda f. \lambda x. f(f(x)) \rightarrow \ast
\]

27 applications of `f`

Using the fact that `addone` has type `f_1:\ast = \hat{\lambda} s. \text{List}(s + 1)(\lambda x. \text{Unsized})`:

\[
f_1 f_1 = \lambda x. f_1(f_1; \ldots; f_1 x; \ldots)
\]

27 applications of `f_1`

We want to prove that this expression is equal to `\hat{\lambda} s. \text{List}(s + 27)(\lambda x. \text{Unsized})`. Because of the partial application now we have to decide the equality of two abstractions. Here the equality is proven by a property that reflects extensionality axiom: `\lambda x. f = \lambda x. g` if and only if `(\lambda x. f)x` and `(\lambda x. g)x` are reduced to the same normal form. To continue with our example we apply fresh variables (e.g. `List a (\lambda y. \text{Unsized})`) to both sides of the equation. For the left hand side we get:

\[
(\lambda x. f_1; f_1; \ldots; f_1 x; \ldots)(\text{List } a (\lambda y. \text{Unsized})) \rightarrow
\]

27 applications of `f_1`

\[
\rightarrow f_1 (f_1; \ldots; (f_1; f_1; \ldots; f_1 x; \ldots; \text{List } a (\lambda y. \text{Unsized})) \ldots) \rightarrow
\]

27 applications of `f_1`

\[
\rightarrow f_1 (f_1; \ldots; \text{List } a + 1 (\lambda x. \text{Unsized}) \ldots) \rightarrow \ast \text{List } a + 27 (\lambda x. \text{Unsized})
\]

26 applications of `f_1`

The following two expressions can be checked similarly:

```plaintext
let t = t3 addone in t3 t : \hat{\lambda} f. \text{List}(s + 9) \lambda y. \text{Unsized}
letfun t (x) : \lambda x. \text{Unsized} = x + 1 in t3 t : \lambda x. \text{Unsized}
```

4 Normalization

Although recursion in size expressions is not allowed, it is easy to express recursive functions using the fixed point combinator `(\text{Y} = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))))`.

\[
\text{fix } (f) : \lambda s. Y s =
\]

let `z = \text{fix } f`

in `f z`

83
Even the size expression is entirely correct it is easy to see that we are not able to check it, because the size expression does not have normal form.

Size expressions are defined as a kind of untyped lambda calculus. The easiest way to reason about normalization and diamond property is to give a typed version of our size expressions and give the rewriting rules from the untyped to the typed version of the size calculus. Our choice of type system is System F. However type inference for System F is generally not possible we will show a way how to construct these types from the underlying type system.

The following function transforms an underlying type \( \tau \) to type of a size expression:

\[
\text{SizeType}(\tau) = \begin{cases} \\
\forall \alpha.\text{SizeType}(a) & \text{if } \tau = \forall \alpha.a \\
\text{SizeType}(a) \rightarrow \text{SizeType}(b) & \text{if } \tau = a \rightarrow b \\
L a & \text{if } \tau = L(a) \\
\tau & \text{if } \tau \text{ is a type variable} \\
\text{Unit} & \text{otherwise}
\end{cases}
\]

We assumed the usual \( \text{Bool} \), \( \text{Nat} \), \( \text{Unit} \) and product types with the usual operations are defined. The following type is also predefined:

\[
L_a := \text{Nat} \times (\text{Nat} \rightarrow a)
\]

This type expresses the fact that a size of a list is a tuple of the length of the list and a map holding the sizes of the elements of the list. The following table gives some examples:

<table>
<thead>
<tr>
<th>Type 1</th>
<th>Type 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Nat} \rightarrow \text{Nat} )</td>
<td>( U \rightarrow U )</td>
<td>( U \rightarrow U )</td>
</tr>
<tr>
<td>( L(\text{Nat}) \rightarrow L(\text{Nat}) )</td>
<td>( L_U \rightarrow L_U )</td>
<td>( L_U \rightarrow L_U )</td>
</tr>
<tr>
<td>( L(a) \rightarrow L(a) )</td>
<td>( \forall a. L_a \rightarrow L_a )</td>
<td>( \forall a. L_a \rightarrow L_a )</td>
</tr>
<tr>
<td>( a \rightarrow b )</td>
<td>( L(a) \rightarrow L(b) )</td>
<td>( \forall a. L_a \rightarrow L_a )</td>
</tr>
<tr>
<td>( (a \rightarrow b) \rightarrow L(a) \rightarrow L(b) )</td>
<td>( \forall a. b : (a \rightarrow b) \rightarrow L_a \rightarrow L_b )</td>
<td>( \forall a. b : (a \rightarrow b) \rightarrow L_a \rightarrow L_b )</td>
</tr>
</tbody>
</table>

### 4.1 Types of the size operators

The \textbf{Unsized} can be easy represented by the \texttt{Unit} type:

\[
\text{Unsized} = \text{unit} : \text{Unit}
\]

\textbf{List} corresponds to the data constructor of a pair:

\[
\text{List} = \Lambda A.\lambda s : \text{Nat}.\lambda f : \text{Nat} \rightarrow A \rightarrow \langle s, f \rangle. \\
: \forall A. \text{Nat} \rightarrow (\text{Nat} \rightarrow A) \rightarrow L_A
\]

If \( \hat{\lambda}f.e \) is seen as a syntactic sugar for \textbf{Unlist} \( AB (\lambda s.f.e) \), where \( A \) and \( B \) are types. On the next subsection we will show that it can be inferred. Now it is easy to describe \( \hat{\lambda}f.e \) with help of the usual projections \( \pi_1 \) and \( \pi_2 \):

\[
\text{Unlist} = \Lambda A.\Lambda B.\lambda f : (\text{Nat} \rightarrow A) \rightarrow B.\lambda t : L_A. f (\pi_1 t) (\pi_2 t) \\
: \forall A. \forall B. (\text{Nat} \rightarrow (\text{Nat} \rightarrow A) \rightarrow B) \rightarrow L_A \rightarrow B
\]
The last thing to do is to define the Shift function:

$$\text{Shift} = \Lambda A. \lambda f : \text{Nat} \to A. \lambda n : \text{Nat} \to A. \lambda g : \text{Nat} \to A. \lambda x : \text{Nat}.
\text{IF} A (x < n) (f x) (g (x - n))$$

$$: \forall (A : \text{Nat} \to A) \to \text{Nat} \to (\text{Nat} \to A) \to \text{Nat} \to A$$

### 4.2 Type inference

It is easy to see that if the underlying type system is a rank-1 predicative type system then all of our types will be rank-1 predicative. It means that the type reconstruction is possible using some kind of Hindley-Milner type inference. As we can tell the correct type of the size expression for any function we need only partial inference, which means it seems it is also possible to check size expressions if the underlying type system is System F using HMF [5] or MLF [4], however investigation of this possibility is a future plan.

### 4.3 \((\hat{\lambda}_f, e_1)e_2\) reduction

We prove that our previously defined reduction rule for \(\hat{\lambda}_f\), and List is correct eg. every size expression which can be typed is strongly normalizable. This can be done by proving that the original reduction rule and the typed reduction gives the same result. So let \(e_1\) and \(e_2\) be fixed expressions and consider the expression \((\hat{\lambda}_f, e_1)e_2\) which is assumed to be well-typed. So there are corresponding \(\tilde{e}_1\) and \(\tilde{e}_2\) sized expressions where \(e_1\) and \(e_2\) can be get by type erasure.

In the untyped system we can assume that \(e_2\) can be reduced to normal form. It must be on the form List \(ab\) because of well-typedness, so we apply our rule, and the result of the application will be \(e_1[s := a][f := b]\).

The most generic type for \(\lambda s. \lambda f. e_1\) is \(\text{Nat} \to (\text{Nat} \to B) \to A\), so the desugared expression is: \((\text{Unlist} \ A \ B \ (\lambda s. \lambda f. \tilde{e}_1)) \tilde{e}_2\). Because of well-typedness the normal form of \(\tilde{e}_2\) is of the form \(<\hat{a}, \hat{b}>\). So the whole expression can be reduced to: \(\lambda s. \lambda f. \tilde{e}_1 \hat{a} \hat{b} \to^{\beta} e_1[s := \hat{a}, f := \hat{b}]\).

Because of the correspondence between typed and untyped calculus \(a\) and \(b\) can be get by type erasure. Hence the result of the untyped expression can be got by type erasure. Taking into account that System F is confluent, our embedding of the rule is sane.

### 4.4 Diamond property of the rewriting system modulo integer ring axiomatics

In this section, instead of \((\text{Shift} u_1 u_2 u_3)u_4\) we consider its desugared definition via branching operator \(\text{IF}\).

Based on commutativity and associativity of addition and multiplication, and their distributivity, we introduce the corresponding equivalence relation on terms of our calculus. It is defined by induction on the term structure:

- \(c \sim c\), \(x \sim x\), \(\text{List} \sim \text{List}\), \(\text{Unsized} \sim \text{Unsized}\),
- \(t_1 \sim t'_1\), \(t_2 \sim t'_2\) \(\Rightarrow t_1 \text{binop} t_2 \sim t'_1 \text{binop} t'_2\),
\(- t_1, t_2 : \text{Nat} \Rightarrow t_1 + t_2 \sim t_2 + t_1, t_1 \ast t_2 \sim t_2 \ast t_1,\)
\(- t_1, t_2, t_3 : \text{Nat} \Rightarrow (t_1 + t_2) + t_3 \sim t_1 + (t_2 + t_3), (t_1 \ast t_2) \ast t_3 \sim t_1 \ast (t_2 \ast t_3),\)
\(- t_1, t_2, t_3 : \text{Nat} \Rightarrow t_1 \ast (t_2 + t_3) \sim t_1 \ast t_2 + t_1 \ast t_3, \) and the inverse distributivity
\(- t_1 \ast t_2 + t_1 \ast t_3 \sim t_1 \ast (t_2 + t_3) \) holds as well,
\(- t \sim t' \Rightarrow \lambda x. t \sim \lambda x. t',\)
\(- t_1 \sim t'_1, t_2 \sim t'_2 \Rightarrow t_1 t_2 \sim t'_1 t'_2,\)
\(- t \sim t' \Rightarrow \lambda x. y. t \sim \lambda x. y. t',\)
\(- t_1 \sim u_1, t_2 \sim u_2, t_3 \sim u_3 \Rightarrow \text{IF}(t_1, t_2, t_3) \sim \text{IF}(u_1, u_2, u_3),\)
\(- \) no other pairs of terms can be added to this relation.

It is an exercise to prove that \(t_1 + (t_2 + t_3) \sim t_1 + (t_2 + t_3)\) and \((t_1 + t_2) \ast t_3 \sim t_1 \ast t_3 + t_2 \ast t_3, \) and \(t_1 \ast t_3 + t_2 \ast t_3 \sim (t_1 + t_2) \ast t_3. \) Moreover, by induction on the
structure of term \(t\) one proves the reflexivity, symmetry and transitivity for \(\sim\) (that is the fact that \(\sim\) is indeed an equivalence). All these statements are proven in Appendix.

Now, we follow the obvious definition of the diamond property modulo \(\sim\) from the paper [6]; the diamond property holds if \(\sim \ast \cdot \sim' \cdot \sim' \ast \sim \ast \sim' ; \) where \(\ast\) denotes composition of relations and \(\sim\) denotes \textit{joinability modulo} \(\sim\) that is the composition \(\sim \ast \sim' \ast \sim\). To prove diamond-modulo-\(\sim\) property for the calculus, we need first to prove a series technical lemmata. Two substitution lemmata above are proven by induction of the structure of terms in the equivalence relation. See Appendix for the full proofs.

\textbf{Lemma 1 (Substitutions 1).} If \(t_1 \sim t_2\) and \(x\) is free in \(t\) then \(t[x := t_1] \sim t[x := t_2].\)

\textbf{Lemma 2 (Substitutions 2).} If \(t \sim t'\) and \(x\) is free in \(t\) and \(t'\) then \(t[x := t''] \sim t'[x := t''].\)

In the next lemma we consider interacting of \(\sim\) with 1-step reduction.

\textbf{Lemma 3 (Reduction).} If \(t_1 \rightarrow t'_1\) and \(t_1 \sim t_2,\) then there exists \(t'_2\) such that \(t_2 \rightarrow t'_2\) and \(t'_1 \sim t_2.\)

\textbf{Proof.} By case of reductions.

\(- \) We start with \(\beta\)-reduction. Let \(t_1 = C_1(\lambda x. t''_1) t''_1 C'_1,\) where \(C_1, C'_1\) are contexts. Then \(t_2 = C_2(\lambda x. t''_2) t''_2 C'_2,\) where \(t''_1 \sim t''_2\) and \(t''_1 \sim t''_2,\) and the corresponding contexts are equivalent. Moreover, then \(t'_1 := C_1 t''_1[x := t''_1] C'_1.\)

We take \(t'_2 = C_2 t''_2[x := t''_2] C'_2 \sim C_2 t''_2[x := t''_2] C'_2 \sim C_2 t''_2[x := t''_2] C'_2 \sim C_1 t''_1[x := t''_1] C'_1 = t'_1\) by the substitution lemmata and the definition of equivalent terms.

\(- \) The case for List-pair reduction is similar to \(\beta\)-reduction and proven by the substitution lemmata as well.

\(- \) Let \(t_1 = C_1(\text{IF True } u_1 u_2) C'_1.\) Then \(t_2 = C_2(\text{IF true } v_1 v_2) C'_2\) for some \(u_i \sim v_i\) with \(i = 1, 2.\) In this case \(t'_1 = u_1 \sim v_1 = t'_2.\) The \textit{False}-case is similar.

Next, by induction on the length of reduction chain, one proves
Lemma 4 (Reduction-Closure). If \( t_1 \rightarrow^* t'_1 \) and \( t_1 \sim t_2 \), then there exists \( t_2' \) such that \( t_2 \rightarrow^* t_2' \) and \( t_1' \sim t_2' \).

Now, we prove the diamond-modulo property.

Lemma 5 (Diamond-Modulo-\( \sim \) Property). \( \sim \cdot \leftarrow^* \rightarrow \sim \subseteq \rightarrow^* \sim \cdot \leftarrow^* \)

Proof. If \( (t, t') \in \sim \cdot \leftarrow^* \rightarrow \sim \cdot \sim \) then there are \( (t'', t''') \in \leftarrow^* \rightarrow \sim \cdot \sim \) such that \( t \sim t'' \) and \( t''' \sim t' \). Since the calculus itself has the diamond property, therefore there exists \( t_1 \) such that \( t'' \rightarrow t_1 \leftarrow t''' \). Using the reduction-closure lemma 4, we obtain that there is \( t_2 \) such that \( t \rightarrow t_2 \) and \( t_2 \sim t_1 \), and there exists \( t_3 \) such that \( t' \rightarrow t_3 \) and \( t_3 \sim t_1 \). From that follows that \( t \rightarrow t_2 \sim t_3 \sim t_1 \sim t' \), that is \( (t, t') \in \rightarrow^* \sim \cdot \sim \cdot \leftarrow^* \)

5 Related work

Structure of size expressions in our research is close to the approach of A. Abel [1], who has applied sized types for termination analysis of higher-order functional programs. For instance, in his notation sized lists of type \( A \) of length \( \iota \) are defined as \( \lambda \iota A.\mu \iota.1 + A \times X \) and size expressions are higher-order arithmetic expressions with \( \lambda \)-abstraction as well. The difference is that in that work one uses linear arithmetic over ordinals, where ordinals represent zero-order sizes. Moreover, in that research size information is not a stand-alone formalism, but a part of dependent-type system.

In the paper [8] the authors go beyond linear arithmetic. For a given higher-order functional program, they obtain a set of first-order arithmetical constraints over unknown cost functions \( f \). Solving these constraints w.r.t. \( f \) gives desired costs of the program. The underlying arithmetic is the arithmetic over naturals, extended with undefined \( \epsilon \) and unbounded \( \omega \) values, equipped with a natural linear order. Size expressions admit addition \( + \), multiplication \( * \) and subtraction of a constant \( -n \), thus such expressions are monotonic. Function types are annotated with natural numbers (latencies), e.g. \( \alpha \rightarrow^* \beta \), so it may be conveniently interpreted as an increment in cost consumption, like \( l \) clock ticks if the resource of interest is time. Our approach is different in a sense that we aim at expressing size dependencies directly in terms of sizes of inputs, bypassing latencies.

In paper [2] the authors approach to complexity analysis of an imperative language, which is a version of Gödel’s T. It is done via abstract interpretation of programs in a semiring of matrices. Informally, matrices represent data flow along program variables. The authors give an upper bound for the return values in term of initial values. However, this is a conjecture and no proof is given. Similarly the conjecture about existing of an abstract interpretation is not proven.

In recent paper [3] the authors develop amortized cost analysis for a higher-order functional language Shopenhauer. The analysis is generic, that is it is applicable to different sorts of resources: heap usage, stack size and the number of function calls. Type-derivation procedure generates linear constraints, solving of which gives desirable upper bounds. The analysis succeeds for sure, if bounds are linear. So far, the methodology does not support polymorphic recursion.
6 Conclusions

We presented a size analysis for higher order functions for a higher-order polymorphic strict functional language. The calculus is based upon the lambda-calculus extending it with arithmetic operations and special operators for finite maps representing size of lists.

We have shown that the extended $\lambda$-calculus we have presented is strongly normalizable (for size expressions of well-typed functions), if a normal form for integer expressions is defined.

We are investigating the possibility to use polynomial interpolation [7] to infer size expressions for higher-order functions as well.

Acknowledgments. The authors would like to convey thanks to Christoph Herrmann for the fruitful discussion.

References

Appendix

Lemma 6 (Associativity 2). $t_1 + (t_2 + t_3) \sim t_1 + (t_2 + t_3)$

Proof. $t_1 + (t_2 + t_3) \sim (t_2 + t_3) + t_1 \sim t_2 + (t_3 + t_1) \sim (t_3 + t_1) + t_2 \sim t_3 + (t_1 + t_2) \sim (t_1 + t_2) + t_3$ due to associativity and commutativity.

Lemma 7 (Distributivity 2). $(t_1 + t_2) * t_3 \sim t_1 * t_3 + t_2 * t_3$ and $t_1 * t_3 + t_2 * t_3 \sim (t_1 + t_2) * t_3$

Proof. – $(t_1 + t_2) * t_3 \sim t_3 * (t_1 + t_2) \sim t_3 * t_1 + t_3 * t_2 \sim t_1 * t_3 + t_2 * t_3$, applying commutativity and distributivity,
– the second equivalence is proved by the symmetric chain.

By induction on the structure of term $t$ we can prove the reflexivity, symmetry and transitivity lemmata (that is the fact that $\sim$ is indeed an equivalence).

Lemma 8 (Reflexivity). $t \sim t$.

Proof. – If $t = x$ is a variable then $t = x \sim x = t$ follows from the assumption and the equivalence $x \sim x$ by the definition.
– Let $t = t_1 \text{binop} t_2$. Then, by induction assumption $t_1 \sim t_1, t_2 \sim t_2$ and by the definition of $\sim$ we have $t = t_1 \text{binop} t_2 \sim t_1 \text{binop} t_2 = t$.
– If $t$ is given by one of the lambda-abstractions or by the application or by $\text{if}$, the proof is similar.

Lemma 9 (Symmetry). $t \sim t' \Rightarrow t' \sim t$.

Proof. From the definition of $\sim$ it follows that $t \sim t'$ must be either an instance of integer axiomatics, or (if not) $t$ and $t'$ must be of the same structure (i.e. either both are variables, or both are composed by $\text{binop}$, or by one of two applications, or by abstractions, or by $\text{if}$).

– Let the equivalence be an instance of the axioms:
  • if $t = (t_1 + t_2)$ for some $t_1, t_2$ and $t \sim t'$ is an instance of commutativity then $t' = (t_2 + t_1)$. Therefore, by the definition of $\sim$ (commutativity case) we have $t' = (t_2 + t_1) \sim (t_1 + t_2) = t$,
  • if $t = ((t_1 + t_2) + t_3)$ for some $t_1, t_2, t_3$ and $t \sim t'$ is an instance of associativity, then $t' = (t_1 + (t_2 + t_3))$. Therefore, by lemma 6 $t' = (t_1 + (t_2 + t_3)) \sim ((t_1 + t_2) + t_3) = t$,
  • if $t = (t_1 + t_2 + t_3)$ for some $t_1, t_2, t_3$ and $t \sim t'$ is an instance of distributivity, then $t' = (t_1 * t_2 + t_1 * t_3)$. Therefore, by the inverse distributivity $t' = (t_1 * t_2 + t_1 * t_3) \sim (t_1 * (t_2 + t_3)) = t$.
  • if $t \sim t'$ is an instance of the inverse distributivity, the proof of $t' \sim t$ is similar to the proof above.
– Let the equivalence do not follow from the axioms. So, both terms in it are of the same structure.
  • If $t = x$ then $t'$ must be $x$ as well, therefore $t' = x \sim x = t$.

89
• Let $t = t_1 \binop t_2$. Therefore $t' = t'_1 \binop t'_2$ as well. Then, by the definition of $\sim$ (and since commutativity is excluded) $t_1 \sim t'_1, t_2 \sim t'_2$. By the induction assumption we have $t'_1 \sim t_1$ and $t'_2 \sim t_2$, therefore $t' = (t'_1 \binop t'_2) \sim (t_1 \binop t_2) = t$.

• If $t$ is given by one of the lambda-abstractions or by the application, or by $\mathsf{If}$, the proof is similar.

Lemma 10 (Transitivity). $t \sim t', t' \sim t'' \Rightarrow t \sim t''$.

Proof. From the definition of $\sim$ it follows that $t \sim t'$ must be either an instance of integer axiomatics, or (if not) $t$ and $t'$ must be of the same structure (i.e. either both are variables, or both are composed by $\binop$, or by one of two applications, or by abstractions, or by $\mathsf{If}$).

- Let the equivalence be an instance of the axioms:
  • if $t = (t_1 + t_2)$ for some $t_1, t_2$ and $t \sim t'$ is an instance of commutativity then $t' = (t_2 + t_1)$. Now, we have to do the same analysis for $t' \sim t''$.
    * if $t' \sim t''$ is an instance of commutativity axiom then $t'' = t_1 + t_2 = t$,
    * if $t' \sim t''$ is an instance of associativity axiom then $t' = (t_2 + t_1)$ with $t_2 = (t_2 + t_2)$ for some $t_2, t_2$. Therefore, $t'' = t_2 + (t_2 + t_1) \sim_t \sim 2 \sim 2 + t_1 \sim 1 + t_1 + t_2 = t$,
  * if $t' \sim t''$ is an instance of distributivity axiom then it may be only an instance of distributivity due to the structure of $t' = (t_2 + t_1)$ and with $t_2 = (t_1 * t_2), t_1 = (t_1 * t_1)$ for some $t_1, t_1, t_2$. Therefore, $t'' = \sim t_1 * (t_2 + t_2) \sim t_1 * t_2 + t_1 * t_1 = t_2 + t_1 \sim t_1 + t_2 = t$,
  * if $t' \sim t''$ is an instance of the structure-cases of the definition of $\sim$, then $t'' = t'_2 + t'_1$, where $t'_2 \sim t_2$ and $t'_1 \sim t_1$; therefore $t'' = t'_2 + t'_1 = t_1 + t_2 = t$.

• the proof is similar (based induction, axiomatics and derived lemmata, the definition of $\sigma$) if the first equivalence is the instance of other axioms.

- Let the first equivalence do not follow from the axioms. So, both terms in it are of the same structure.
  • If $t = x$ then $t'$ must be $x$ as well and the same holds for $t''$, therefore $t'' = x \sim x = t$.
  • If the second equivalence is an instance of axioms, then the proof is similar to the proof for the pair of equivalences with the axiom being first and the structural case being second, see above.
  • If both equivalences are given by a structural case of the definition of $\sim$, then the proof is straightforward by induction assumption.

Lemma 11 (Substitutions 1). If $t_1 \sim t_2$ and $x$ is free in $t$ then $t[x := t_1] \sim t[x := t_2]$.

Proof. – If $t = x$ is a variable then $t[x := t_1] = t_1 \sim t_2 = t[x := t_2]$ follows from the assumption $t_1 \sim t_2$.

– Let $t = (t' \binop t'')$. Then $t[x := t_1] = (t'[x := t_1] \binop t''[x := t_1]) \sim (t'[x := t_2] \binop t''[x := t_2]) = t[x := t_2]$ by the induction assumption.
If $t$ is given by one of the lambda-abstractions or by the application or by $\text{If}$, the proof is similar.

**Lemma 12 (Substitutions 2).** If $t \sim t'$ and $x$ is free in $t$ and $t'$ then $t[x := t''] \sim t'[x := t''].$

**Proof.** From the definition of $\sim$ it follows that $t \sim t'$ must be either an instance of integer axiomatics, or (if not) $t$ and $t'$ must be of the same structure (i.e. either both are variables, or both are composed by \text{binop}, or by one of two applications, or by abstractions, or by $\text{If}$).

Let the equivalence be an instance of the axioms:

- If $t = (t_1 + t_2)$ for some $t_1, t_2$ and $t \sim t'$ is an instance of commutativity then $t' = (t_2 + t_1)$. Therefore, by the definition of $\sim$ (commutativity case)
we have $t[x := t''] = (t_1[x := t''] + t_2[x := t'']) \sim (t_2[x := t''] + t_1[x := t'']) = t'[x := t''].$

- If $t = (t_1 + (t_2 + t_3))$ for some $t_1, t_2, t_3$ and $t \sim t'$ is an instance of associativity, then $t' = (t_1 + (t_2 + t_3))$. Therefore, by the associativity
property $t[x := t''] = ((t_1[x := t''] + t_2[x := t'']) + t_3[x := t'']) \sim (t_1[x := t''] + t_2[x := t''] + t_3[x := t'']) = t'[x := t''].$

- If $t = (t_1 * (t_2 + t_3))$ for some $t_1, t_2, t_3$ and $t \sim t'$ is an instance of distributivity, then $t[x := t''] = (t_1[x := t''] * (t_2[x := t''] + t_3[x := t''])) \sim (t_1[x := t''] * t_2[x := t''] + t_1[x := t''] * t_3[x := t'']) = (t_1 * t_2 + t_1 * t_3)[x := t''] = t'[x := t''].$

- If $t \sim t'$ is an instance of the inverse distributivity, the proof of $t' \sim t$ is similar to the proof above.

Let the equivalence do not follow from the axioms. So, they are of the same structure.

- If $t = x$ then $t' = x$ and $t'[x := t''] = [x := t''] = t'' \sim t'' = [x := t''] = t''[x := t''].$

- Let $t = t_1 \text{binop} t_2$. Then $t' = t_1' \text{binop} t_2'$ for some $t_1', t_2'$. Further, $t[x := t''] = (t_1[x := t''] \text{binop} t_2[x := t'']) \sim (t_1'[x := t''] \text{binop} t_2'[x := t'']) = t'[x := t''].$

- If $t$ is given by one of the lambda-abstractions or by the application, or by $\text{If}$, the proof is similar.
Refinement Types for Pattern-match Analysis

Stefan Holdermans¹ and Jurriaan Hage²

¹ Vector Fabrics
Paradijslaan 28, 5611 KN Eindhoven, The Netherlands
stefan@vectorfabrics.com

² Department of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB, Utrecht, The Netherlands
jur@cs.uu.nl

Abstract. Functional programming languages like Haskell and ML allow for partial functions to be defined by means of nonexhaustive pattern matches. Injudicious use of such functions may lead to run-time failures that are often hard to debug. In this extended, we sketch a system of refinement types for keeping track of values that are subjected to nonexhaustive pattern matches. The imposed typing discipline facilitates pinpointing at compile time a large class of suspicious applications of partial functions and, consequently, prevents programmers from introducing subtle bugs due to broken shape invariants.

Key words: type-and-effect systems, program verification, pre- and postconditions

1 Introduction

Every Haskell programmer knows what it is like to write a program, compile it, run it, and then to experience something similar to:

```
$ ./a.out
a.out: Prelude.head: empty list
```

That is, to find out that, at some point in the execution of the program, an attempt was made to paradoxically retrieve the first element of an empty list, causing the program to abort with a run-time failure.

Those who prefer Caml (or, for that matter, an other dialect of ML) are in no better shape:

```
$ ./b.out
Fatal error: exception Failure("hd")
```

It is the authors’ experience that bugs like these—which we like to refer to as headaches—typically reveal themselves just one or two days shy of an important deadline.

Below we discuss the foundations of a type-and-effect system that allows for detecting headaches at compile time by identifying problematic invocations of nonexhausting pattern matches. In particular, the contributions of our work are the following:
We define a semantics-based program analysis for keeping track of which values are being subjected to pattern matches in functional programs. The analysis is presented as a type-and-effect system [1] in which so-called refinement types give a precise account of the shapes of values and effects convey whether or not a computation may result in a run-time error. In particular, the system rejects pattern matches that are nonexhaustive with respect to the shapes of the values they are applied to.

We discuss how refinement polymorphism facilitates polyvariance and, hence, a more precise and modular analysis.

We discuss how a precise account for nonstrict evaluation affects the look and feel of our type system.

We have a prototype implementation of our analysis for a representative subset of Haskell 98 [2] and plan to extend it to a pattern-match analyser that covers the complete language and that is to be made available as an open-source tool.

In this preliminary paper, we only highlight the key ideas that underlie our analysis. The details of our work are deferred to the forthcoming full research article.

2 Motivation

Headaches in functional programs are due to applications of partial functions that are defined in terms of nonexhaustive pattern matches. Bugs that are introduced by such nonexhaustive pattern matches are typically hard to debug: often, the function that contains the nonexhaustive pattern match was defined to be partial by intention and blame should be assigned to the caller of the function. Execution traces can reveal the sequence of function calls that eventually led to an invocation of the partial function, but do not show which calls were responsible for producing the argument to that function.

Compilers typically acknowledge the problematic nature of nonexhaustive matches and are capable of producing warning messages when a potentially nonexhaustive pattern match is defined. However, as we mentioned above, nonexhaustive pattern matches may be used intentionally; compiler messages that warn for nonexhaustive matches are then easily suppressed by extending an incomplete match with a catch-all case that immediately raises a run-time error.

Our solution to the problem consists of a pattern-match analysis that allows for the definition of nonexhaustive pattern matches, but that rejects invocations of a incomplete match on values that have shapes that are not covered by the cases of the match.

3 Key Ideas

Our approach to pattern-match analysis centers around a type-and-effect system that assigns to each valid expression in a functional program a refined type, a
refinement, and an effect. Valid expressions are those that are well-typed in a traditional sense as well as “safe” with respect to nonexhaustive pattern matches.

A refined type is a type expression in which the argument and result positions of function types are decorated with refinements. Refinements are essentially sets of constructor applications, conveying the shapes of the values to which an expression may reduce. For example, if it can be shown that a Boolean expression $e$ will always reduce to True, then $e$ can be assigned the type Bool together with the refinement $\{\text{True}\}$, written

$$e::\text{Bool}@\{\text{True}\}.$$ 

If both False and True are possible results of evaluating $e$, then $e$ will be assigned the refinement $\{\text{False, True}\}$, as in

$$e::\text{Bool}@\{\text{False, True}\}.$$ 

In refinements, nonnullary constructors are applied to arguments that are themselves refinements as well. For example, a list $[\text{False, True}]$ may be typed as

$$[\text{False, True}]:=[\text{Bool}](\{\text{False}\}\{\text{True}\}\{[]\}).$$

In this particular example, the refinement encodes the length of the list. In general, of course, it may not be possible to derive such precise shapes for expressions. To this end, the refinement language includes a fixed-point construct to denote recursive refinements. For instance, to denote the possible shapes of a list of unknown length containing Booleans, we can assign the refinement

$$\mu b.\{[],(:)\{\text{False, True}\}\} b.$$ 

Here, $b$ is a variable ranging over refinements: its occurrence as the second argument to $(:)$ refers to the refinement itself.

Consider now the function head,

$$\text{head} (x:_-) = x.$$ 

As a function, it receives the refinement $\lambda$, indicating that it produces a lambda-constructed value. But what type will it have assigned? Recall, that refined function types have their argument and result positions decorated with refinements. The idea is that the decoration on the result type conveys the possible shapes that the result can take. The decoration on the argument type expresses a precondition for the function’s arguments: to be precise, it expresses for which argument shapes the function is guaranteed to produce a nondiverging result. Hence, for head we have

$$\text{head} :: \forall a b c. (([a]@\{\text{b c}\}) \to (a@b))@\{\lambda\},$$ 

with $a$ ranging over refined types and $b$ and $c$ ranging over refinements.
How our analysis helps programmers avoiding headaches becomes clear when we apply a partial function like \texttt{head} to an argument. We have, for example,

\[
\text{head [False, True]} :: \text{Bool}\{\text{False}\},
\]

but \texttt{head [\_]} with \texttt{[]} :: \forall a. [a]@{} cannot be typed as the refinement for the argument does not satisfy an instantiation of the precondition \{(\_): b@{}\} of \texttt{head}. In general, an argument refinement satisfy the precondition in a function type if the shapes expressed by the former refinement form a subset of the shapes expressed by the latter. If, for a given function application, the argument refinement does not satisfy the precondition, an implementation of our analysis will produce an error message.

Often, partial functions like \texttt{head} are in fact defined in terms of what, strictly speaking, is an exhaustive pattern match; for example, as in

\[
\text{head (x : \_)} = x \\
\text{head \_} = \text{error "Prelude.head: empty list"}.
\]

As there is no essential difference with the previous definition—that is, \texttt{head} still diverges on the empty list—we still want to consider this pattern match as nonexhaustive and have our analysis produce a precondition that expresses that the argument to \texttt{head} should not be empty. To this end, we compute for each expression, in addition to a type and a refinement, an effect that indicates whether or not evaluation of the expression may diverge. We write \(\bot\) for the effect of an expression that is guaranteed to be nondiverging and \(\top\) for the effect of an expression that may diverge. Then we have

\[
\text{error "Prelude.head: empty list"} :: \forall a. a@{} & \top,
\]

indicating that a call to \texttt{error} will not produce a value and should be expected to diverge. Now, when analysing pattern matches, we exclude alternatives with right-hand sides that carry the effect \(\top\) from contributing to preconditions. This way, the definition of \texttt{head} in terms of \texttt{error} still receives an argument refinement of the form \{(\_): b@{}\}.

Another notorious example of a partial function defined in terms of a nonexhaustive pattern match is the function \texttt{fromJust} that attempts to extract an element from a \texttt{Maybe}-value:

\[
\text{fromJust (Just x)} = x \\
\text{fromJust Nothing} = \text{error "Maybe.fromJust: Nothing"}.
\]

Here we have

\[
\text{fromJust} :: \forall b. ((\text{Maybe } a@{}\{\text{Just } b\}) \rightarrow (a@b))@{} & \bot.
\]

Compare this to

\[
\text{fromMaybe \_ (Just x)} = x
\]
fromMaybe x Nothing = x

with

fromMaybe :: ∀a b c.
((a@b) → ((Maybe a@{Just c,Nothing}) → (a@(b ∪ c)))@{λ})@{λ} & ⊥ .

Here is an example of a recursive function with a nonexhaustive pattern match:

last [x] = x
last ((_:xs)) = last xs .

For this function, pattern-match analysis results in

last :: ∀a b c.(((a@µb. {Cons c {[],b}}) → (a@c))@{λ}) & ⊥ .

Finally, consider the analysis of higher-order functions like

map f [] = []
map f (x:xs) = f x : map f xs ,

with

map :: ∀a b c d.
((((a@b) → (c@d))@{λ}) →
(([[a]@µe. {[],(:) b e}) → ([c]@µe. {[],(:) d e}))@{λ}))@{λ} & ⊥ ,

such that

map head :: ∀a b c.
(([[a]@µd. {[],(:) b c} d}) → ([a]@µd. {[],(:) b d}))@{λ} & ⊥ .

This last type expresses, as expected, that map head is defined on any list containing nonempty lists as elements.

A Note on Polymorphism. In the examples above, we have used universally quantified variables to assign refinement types that are polymorphic in some of their refinements. It is well-known that, in type-and-effect systems, parametric polymorphism in its simplest form allows for analyses that are modular but not necessarily very precise. To improve on their precision, one could introduce some form of bounded polymorphism—for example by means of so-called qualified types [3, 4]—or extend the refinement language so that it allows for more involved operations on refinements, somewhat in the spirit of type families [5]. These techniques have the advantage that they have been implemented in the context of Haskell’s type classes and, hence, a static pattern-match analyser for Haskell is likely to be able to take advantage of already available infrastructure.

An alternative approach to refinement polymorphism is to use so-called intersection types [6, 7].
4 Related Work

Catch. To those who know their way around in the ever so flourishing Haskell community it will hardly come as a surprise that our work on pattern-match analysis is heavily inspired by Neil Mitchell’s Catch\(^3\). A static partiality checker for Haskell 98, Catch performs whole-program analysis on its input, spotting all nonexhaustive pattern matches in a program. It then generates, for each function, a precondition that callers should meet in order to avoid run-time pattern-match errors. If all function calls in a program satisfy the preconditions of their callees, the program is considered “safe”.

Under the hood, Catch makes use of a constraint-based analysis that is defined in a somewhat ad-hoc fashion [8–10], its specification mainly algorithmic and closely tied to the underlying implementation in Haskell. Our analysis, using the idiom of type-and-effect systems, is modular and can be defined largely independent from its implementation.

An important limitation of Catch—and a major difference with our analysis—is its restriction to first-order functions. Programs that contain higher-order functions first need to go through a preprocessing phase in which the program is defunctionalised [11].

Other related work. Other related work includes Dana Xu’s work on static contract checking which allows for the explicit declaration of pre- and postconditions for functions [12–14]. Furthermore, our type refinements are reminiscent of the types used in soft typing for dynamic languages [15].

5 Conclusion

We have briefly discussed the key ideas that underlie our type-based pattern-match analysis. Further details—such as the actual typing rules, the impact of nonstrict evaluation on the definition of the analysis, precision, and modularity—are to be provided in the full paper.

References


\(^3\) See http://community.haskell.org/~ndm/catch.


Abstract. Compilers for functional languages, whether strict or non-strict, typed or untyped, need to handle many of the same problems, for example thunks, lambda lifting, optimisation, garbage collection, and system interaction. Although implementation techniques are by now well understood, it remains difficult for a new functional language to exploit these techniques without either implementing a compiler from scratch, or attempting fit the new language around another existing compiler. Epic is a compiled functional language which exposes functional compilation techniques to a language implementor, with a Haskell API. In this paper we describe Epic and outline how it may be used to implement a high level language compiler, illustrating our approach using a dynamically typed graphics language.

1 Introduction

When implementing a new language, whether for research purposes or as a realistic general purpose language, we are inevitably faced with the problem of executing the language. Ideally, we would like execution to be as fast as possible, and exploit known techniques from many years of compiler research. However, it is difficult to make use of the existing available back ends for functional languages, such as the STG [10, 12, 14] or ABC [13] machines. They may be too low level, they may make assumptions about the source language (e.g. its type system) or there may simply be no clearly defined API. As a result, experimental languages such as Agda [11] have resorted to generating Haskell, using unsafeCoerce to bypass the type system. Similarly, Cayenne [1] used LML with the type checker switched off. This is not ideal for several reasons: we cannot expect to use the full power and optimisations of the underlying compiler, nor can we expect it to exploit any specific features of our new source language, such as the optimisation opportunities presented by rich dependent type systems.

Epic aims to provide the necessary features for implementing the back-end of a functional language — thunks, closures, algebraic data types, scope management, lambda lifting — without imposing any design choices on the high level language designer, with the obvious exception that a functional style is encouraged! The library provides compiler combinators, which guarantee that any output code will be syntactically correct and well-scoped. This gives a simple
method for building a compiler for a new functional language, e.g. for experimentation with new type systems or new domain specific languages.

Epic was originally written as a back end for Epigram [6] (the name\(^1\) is short for “Epigram Compiler”). It is now used by Idris [3] and as an experimental back end for Agda. It is specifically designed for reuse by other source languages.

2 The Epic Language

Epic is based on the \(\lambda\)-calculus with some extensions. It supports primitives such as strings and integers, as well as tagged unions. There are additional control structures for specifying evaluation order, primitive loop constructs, and calling foreign functions. Foreign function calls are annotated with types, to assist with marshaling values between Epic and C, but otherwise there are no type annotations and there is no type checking — as Epic is intended as an intermediate language, it is assumed that the high level language has already performed any necessary type checking. The abstract syntax of the core language is given in Figure 1. As a shorthand, we use de Bruijn telescope notation, \(\vec{x}\), to denote a sequence of \(x\). We use \(x\) to stand for variable names, and \(i\), \(b\), \(f\), \(c\), and \(str\) to stand for integer, boolean, floating point, character and string literals respectively.

2.1 Definitions

An Epic program consists of a sequence of untyped function definitions, with zero or more arguments. The entry point is the function \(\text{main}\), which takes no arguments. For example:

\[
\begin{align*}
\text{factorial} & \quad = \quad \text{if} \quad x \quad == \quad 0 \quad \text{then} \quad 1 \\
& \quad \quad \quad \quad \quad \text{else} \quad x \quad \times \quad \text{factorial}(x - 1)
\end{align*}
\]

\[
\begin{align*}
\text{main} & \quad = \quad \text{putStrLn}(\text{intToString}(\text{factorial}(10)))
\end{align*}
\]

The right hand side of a definition is an expression consisting of function applications, operators (arithmetic, comparison, and bit-shifting), bindings and control structures (some low level and imperative). Functions may be partially applied.

Values Values in an Epic program are either one of the primitives (an integer, floating point number, character, boolean or string) or a tagged union. Tagged unions are of the form \(\text{Con}\_i(t_1, \ldots, t_n)\), where \(i\) is the tag and the \(t_i\) are the fields. The name \(\text{Con}\) is to suggest “Constructor”. For example, we could represent a list using tagged unions, with \(\text{Con}0()\) representing the empty list and \(\text{Con}1(x, xs)\) representing a cons cell, where \(x\) is the element and \(xs\) is the tail of the list.

Tagged unions are inspected either using field projection \((t!i)\) projects the \(i\)th field from a tagged union \(t)\) or by case analysis. e.g., to append two lists:

\(^1\) Coined by James McKinna
\[ p ::= \text{def} \quad \text{(Epic program)} \quad \text{def} ::= x(\vec{x}) = t \quad \text{(Definition)} \]

\[ t ::= x \quad \text{(Variable)} \quad | \quad t(\vec{t}) \quad \text{(Application)} \]
\[ | \quad \lambda x. t \quad \text{(Lambda binding)} \quad | \quad \text{let } x = t \text{ in } t \quad \text{(Let binding)} \]
\[ | \quad \text{Con } i(\vec{t}) \quad \text{(Constructor)} \quad | \quad t!i \quad \text{(Projection)} \]
\[ | \quad t \text{ op } t \quad \text{(Infix operator)} \quad | \quad \text{if } t \text{ then } t \text{ else } t \quad \text{(Conditional)} \]
\[ | \quad \text{case } t \text{ of } \vec{alt} \quad \text{(Case expressions)} \quad | \quad \text{lazy}(t) \quad \text{(Lazy evaluation)} \]
\[ | \quad \text{effect}(t) \quad \text{(Effectful term)} \quad | \quad \text{while } t \text{ t} \quad \text{(While loops)} \]
\[ | \quad x := t \text{ in } t \quad \text{(Variable update)} \quad | \quad \text{foreign } T \text{ str } (t : T) \quad \text{(Foreign call)} \]
\[ | \quad \text{malloc } t \text{ t} \quad \text{(Allocation)} \quad | \quad i \mid f \mid c \mid b \mid \text{str} \quad \text{(Constants)} \]

\[ \vec{alt} ::= \text{Con } i(\vec{x}) \mapsto t \quad \text{(Constructors)} \quad | \quad i \mapsto t \quad \text{(Constants)} \]
\[ | \quad \text{default } \mapsto t \quad \text{(Match anything)} \]

\[ \text{op ::= + | − | × | / | == | < | ≤ | > | ≥ | << | >>} \]

\[ T ::= \text{Int} \mid \text{Char} \mid \text{Bool} \mid \text{Float} \mid \text{String} \quad \text{(Primitives)} \]
\[ | \quad \text{Unit} \quad \text{(Unit type)} \]
\[ | \quad \text{Ptr} \quad \text{(Foreign pointers)} \]
\[ | \quad \text{Any} \quad \text{(Polymorphic type)} \]

\[ \text{Fig. 1. Epic syntax} \]

\[ \text{append}(xs, ys) = \text{case } xs \text{ of} \]
\[ \quad \text{Con } 0() \mapsto ys \]
\[ \quad \text{Con } 1(x, xs') \mapsto \text{Con } 1(x, \text{append}(xs', ys)) \]

**Evaluation Strategy** By default, expressions are evaluated eagerly (in applicative order), i.e. arguments to functions and tagged unions are evaluated immediately, left to right. Evaluation can instead be delayed using the lazy construct. An expression \text{lazy}(t) will not be evaluated until it is required by one of:

- Inspection in a \text{case} expression or the condition in an \text{if} statement.
- Field projection.
- Being passed to a foreign function.
- Explicit evaluation with \text{effect}. This evaluates side-effecting code (and does not update the thunk).

**Higher order functions** Finally, expressions may contain \text{λ} and \text{let} bindings. Higher order functions such as \text{map} are also permitted:
\[
map(f, xs) = \text{case } xs \text{ of} \\
\quad \text{Con } 0() \mapsto \text{Con } 0() \\
\quad \text{Con } 1(x, xs') \mapsto \text{Con } 1(f(x), map(f, xs'))
\]

\[
evens(n) = \text{let } \text{nums} = \text{take}(n, \text{countFrom}(1)) \text{ in} \\
\quad \text{map}(\lambda x. x \times 2, \text{nums})
\]

### 2.2 Foreign Functions

Most programs eventually need to interact with the operating system. Epic provides a lightweight foreign function interface which allows interaction with external C code. Since Epic does no type checking or inference, a foreign call requires the argument and return types to be given explicitly, e.g. the C function:

```c
double sin(double x);
```

We can call this function from Epic by giving the C name, the return type (an Epic Float) and the argument type (also an Epic Float).

```haskell
sin(x) = foreign Float "sin" (x : Float)
```

### 2.3 Low Level Features

Epic emphasises control over safety, and therefore provides some low level features to give language implementations more control over generated code. A high level language may wish to use these features in some performance critical contexts, whether for sequencing side effects, implementing optimisations, or to provide run-time support code. Epic allows sequencing, while loops and variable update, and provides a `malloc` construct for memory allocation. The behaviour of `malloc n t` is to create a fixed pool of \(n\) bytes, and allocate only from this pool when evaluating \(t\).

### 2.4 Haskell API

The primary interface to Epic is through a Haskell API, which is used to build expressions and programs, and to compile them to executables. Implementing a compiler for a high level language is then a matter of converting the abstract syntax of a high level program into an Epic program, through these “compiler combinators”, and implementing any run-time support as Epic functions.

**Programs and expressions** The API allows the building of Epic programs with an Embedded Domain Specific Language (EDSL) style interface, i.e. we try to exploit Haskell’s syntax as far as possible. There are several possible representations of Epic expressions. `Expr` is the internal abstract representation, and `Term` is a representation which carries a name supply. We have a type class
EpicExpr which provides a function term for building a concrete expression using a name supply:

```haskell
type Term = State Int Expr

class EpicExpr e where
term :: e -> Term
```

There are straightforward instances of EpicExpr for the internal representations Expr and Term. There is also an instance for String, which parses concrete syntax, which is beyond the scope of this paper. More interestingly, we can build an instance of the type class which allows Haskell functions to be used to build Epic functions. This means we can use Haskell names for Epic references, and not need to worry about scoping or ambiguous name choices.

```haskell
instance EpicExpr e => EpicExpr (Expr -> e) where
```

If we have explicit names for arguments, it can be more convenient to pass these to Epic directly rather than to use the name supply. We provide an instance to allow a user to give argument names explicitly:

```haskell
instance EpicExpr e => EpicExpr ([Name], e) where
```

Both forms, using Haskell functions or explicit names, can be mixed freely in an expression. A program is a collection of named Epic expressions built using the EpicExpr class:

```haskell
type Program = [EpicDecl]

data EpicDecl = forall e. EpicExpr e => EpicFn Name e
```

The library provides a number of built-in definitions for some common operations such as outputting strings and converting data types:

```haskell
basic_defs :: [EpicDecl]
```

In this paper we use `putStr` and `putStrLn` for outputting strings, `append` for concatenating strings, and `intToString` for integer to string conversion. We can compile a collection of definitions to an executable, or simply execute them directly. Execution begins with the function called "main" — Epic reports an error if this function is not defined:

```haskell
compile :: Program -> FilePath -> IO ()
run :: Program -> IO ()
```

**Building expressions** We have seen how to build λ bindings with the EpicExpr class, using either Haskell’s λ or pairing explicitly bound names with their scope. We now add further sub-expressions. Firstly, we introduce functions for referring to explicitly bound names — `ref` build a reference to a Name, `name` converts a string into the internal form, and `fn` combines these operations.
ref :: Name → Term
name :: String → Name
fn :: String → Term

We would refer to the putStrLn function, for example, with fn "putStrLn". The general form of functions which build expressions is to create a Term — i.e. an expression which manages its own name supply — by combining arbitrary Epic expressions (i.e. instances of EpicExpr). For example, to apply a function to an argument, we provide an EpicExpr for the function and the argument:

```
infixl 5 @@
(@@) :: (EpicExpr f, EpicExpr a) => f → a → Term
```

Since Term itself is an instance of EpicExpr, we can apply a function to several arguments through nested applications of @@, which associates to the left as with normal Haskell function application. We have several arithmetic operators, including arithmetic, comparison and bitwise operators, e.g.:

```
plus_, minus_, times_, divide_, :: Op
```

We follow the convention that Epic keywords and primitive operators are represented by a Haskell function with an underscore suffix. This convention arises because we cannot use Haskell keywords such as if, let and case as function names. For consistency, we have extended the convention to all functions and operators. if...then...else expressions are built using the if_ function:

```
if_ :: (EpicExpr a, EpicExpr t, EpicExpr e) => a → t → e → Term
```

For let bindings, we can either use higher order syntax or bind an explicit name. To achieve this, as before, we implement a type class and instances which support both:

```
class LetExpr e where
  let_ :: EpicExpr val ⇒ val → e → State Int Expr

instance EpicExpr sc ⇒ LetExpr (Name, sc)
instance       LetExpr (Expr → Term)
```

To build a constructor form, we apply a constructor with an integer tag to its arguments. We build a constructor using the con_ function, and provide a shorthand tuple_ for the common case where the tag can be ignored — as the name suggests, this happens when building tuples and records:

```
con_ :: Int → Term
tuple_ :: Term
```

**Case analysis** To inspect constructor forms, or to deconstruct tuples, we use case expressions. A case expression chooses one of the alternative executions path depending on the value of the scrutinee, which can be any Epic expression:
We leave the definition of `Case` abstract (although it is simply an Epic expression carrying a name supply) and provide an interface for building case branches. The scrutinee is matched against each branch, in order. To match against a constructor form, we use the same trick as we did for λ-bindings, either allowing Haskell to manage to scope of constructor arguments, or giving names explicitly to arguments, or a mixture:

```haskell
class Alternative e where
    mkAlt :: Tag -> e -> Case

instance Alternative Expr
instance Alternative Term

instance Alternative e => Alternative (Expr -> e)
instance Alternative e => Alternative ([Name], e)
```

We can build case alternatives for constructor forms (matching a specific tag), tuples, or integer constants (matching a specific constant), as well as a default case if all other alternatives fail to match. In each of the following, `e` is an expression which gives the argument bindings, if any, and the right hand side of the match.

```haskell
con :: Alternative e => Int -> e -> Case
tuple :: Alternative e => e -> Case
constcase :: EpicExpr e => Int -> e -> Case
defaultcase :: EpicExpr e => e -> Case
```

### 3 Example — Compiling the λ-Calculus

In this section we present a compiler for a simple high level language. This is a compiler for the untyped λ-calculus using Higher Order Abstract Syntax, which shows the fundamental features of Epic required to implement a complete compiler. We have also implemented compilers for λΠ [9], a dependently typed language, which shows how Epic can handle languages with more expressive type systems, and a dynamically typed graphics language\(^2\), which shows how Epic can be used for languages with run-time type checking and which require foreign function calls.

#### 3.1 Representation

Our example is an implementation of the untyped λ-calculus, plus primitive integers and strings, and arithmetic and string operators. The language is represented in Haskell using higher order abstract syntax (HOAS). That is, we

\(^2\) [http://hackage.haskell.org/package/atuin](http://hackage.haskell.org/package/atuin)
represent λ-bindings \((\text{Lam})\) as Haskell functions, using a Haskell variable name to refer to the locally bound variable. We also include global references \((\text{Ref})\) which refer to top level functions, function application \((\text{App})\), constants \((\text{Const})\) and binary operators \((\text{Op})\):

\[
\text{data Lang} = \text{Lam} \ (\text{Lang} \rightarrow \text{Lang}) \\
| \text{Ref} \ \text{Name} \\
| \text{App} \ \text{Lang} \ \text{Lang} \\
| \text{Const} \ \text{Const} \\
| \text{Op} \ \text{Infix} \ \text{Lang} \ \text{Lang}
\]

Constants can be either integers or strings:

\[
\text{data Const} = \text{CInt} \ \text{Int} \ |
\text{CStr} \ \text{String}
\]

There are infix operators for arithmetic \((\text{Plus}, \text{Minus}, \text{Times}, \text{Divide})\), string manipulation \((\text{Append})\) and comparison \((\text{Eq}, \text{Lt}, \text{Gt})\). The comparison operators return an integer — zero if the comparison is true, non-zero otherwise:

\[
\text{data Infix} = \text{Plus} \ |
\text{Minus} \ |
\text{Times} \ |
\text{Divide} \ |
\text{Append} \ |
\text{Eq} \ |
\text{Lt} \ |
\text{Gt}
\]

A complete program consists of a collection of named \text{Lang} definitions:

\[
\text{type Defs} = [(\text{Name}, \text{Lang})]
\]

### 3.2 Compilation

Our aim is to convert a collection of \text{Defs} into an executable, using the \text{compile} or \text{run} function from the Epic API. Given an Epic \text{Program}, \text{compile} will generate an executable, and \text{run} will generate an executable then run it. Recall that a program is a collection of named Epic declarations:

\[
\text{data EpicDecl} = \forall e. \ \text{EpicExpr} \ e \Rightarrow \text{EpicFn} \ \text{Name} \ e \\
| \ldots
\]

\[
\text{type Program} = [\text{EpicDecl}]
\]

Our goal is to convert a \text{Lang} definition into something which is an instance of \text{EpicExpr}. We use \text{Term}, which is an Epic expression which carries a name supply. Most of the term construction functions in the Epic API return a \text{Term}.

\[
\text{build} :: \text{Lang} \rightarrow \text{Term}
\]

The full implementation of \text{build} is given in Figure 2. In general, this is a straightforward traversal of the \text{Lang} program, converting \text{Lang} constants to Epic constants, \text{Lang} application to Epic application, and \text{Lang} operators to the appropriate built-in Epic operators.

The cases worth noting are the compilation of λ-bindings and string concatenation. Using HOAS has the advantage that Haskell can manage scoping, but the disadvantage that it is not straightforward to convert the abstract syntax into another form. The Epic API also allows scope management using HOAS, so
build :: Lang -> Term
build (Lam f) = term (\x -> build (f (EpicRef x)))
build (EpicRef x) = term x
build (Ref n) = ref n
build (App f a) = build f @@ build a
build (Const (CInt x)) = int x
build (Const (CStr x)) = str x
build (Op Append l r) = fn "append" @@ build l @@ build r
build (Op op l r) = op_ (eOp op) (build l) (build r)
  where eOp Plus = plus_
        eOp Minus = minus_
        ...

Fig. 2. Compiling Untyped λ-calculus

we need to convert a function where the bound name refers to a Lang value into a function where the bound name refers to an Epic value. The easiest solution is to extend the Lang datatype with an Epic reference:

data Lang = ...
  | EpicRef Expr

build (Lam f) = term (\x -> build (f (EpicRef x)))

To convert a Lang function to an Epic function, we build an Epic function in which we apply the Lang function to the Epic reference for its argument. Every reference to a name in Lang is converted to the equivalent reference to the name in Epic. While there may be neater solutions involving an environment, or avoiding HOAS, this solution is very simple to implement, and preserves the desirable feature that Haskell manages scope. Compiling string append uses a built in function provided by the Epic interface in basic_defs:

build (Op Append l r) = fn "append" @@ build l @@ build r

Given build, we can translate a collection of HOAS definitions into an Epic program, add the built-in Epic definitions and execute it directly. Recall that there must be a function called "main" or Epic will report an error.

mkProgram :: Defs -> Program
mkProgram ds = basic_defs ++
  map (\ (n, d) -> EpicFn n (build d)) ds

execute :: Defs -> IO ()
execute p = run (mkProgram p)

Alternatively, we can generate an executable. Again, the entry point is the Epic function called "main":

comp :: Defs -> IO ()
comp p = compile "a.out" (mkProgram p)
This is a compiler for a very simple language, but a compiler for any more complex language follows the same pattern: convert the abstract syntax for each named definition into a named Epic Term, add any required primitives (we have just used basic_defs here), and pass the collection of definitions to run or compile.

4 Atuin — A Dynamically Typed Graphics Language

In this section we present a more detailed example language, Atuin\(^3\), and outline how to use Epic to implement a compiler for it. Atuin is a simple imperative language with higher order procedures and dynamic type checking, with primitive operations implementing turtle graphics. The following example illustrates the basic features of the language. The procedure repeat executes a code block a given number of times:

\[
\text{repeat}(\text{num}, \text{block}) \{
\quad \text{if } \text{num} > 0 \{
\quad \quad \text{eval block}
\quad \quad \text{repeat}(\text{num}-1, \text{block})
\quad \}
\}\]

Using repeat, polygon draws a polygon with the given number of sides, a size and a colour:

\[
\text{polygon}(\text{sides}, \text{size}, \text{col}) \{
\quad \text{if } \text{sides} > 2 \{
\quad \quad \text{colour col}
\quad \quad \text{angle} = 360/\text{sides}
\quad \quad \text{repeat}(\text{sides}, \{
\quad \quad \quad \text{forward size}
\quad \quad \quad \text{right angle}
\quad \quad \})
\quad \}
\}\]

Programs consist of a number of procedure definitions, one of which must be called main and take no arguments:

\[
\text{main()} \{
\quad \text{polygon}(10,25,\text{red})
\}\]

4.1 Abstract Syntax

In this section we discuss the abstract syntax of Atuin, as algebraic data types constructed by a Happy-generated parser. Constants can be one of four types: integers, characters, booleans and colours:

\(^3\) http://hackage.haskell.org/package/atuin
data Const = MkInt Int
    | MkChar Char
    | MkBool Bool
    | MkCol Colour

data Colour = Black | Red | Green | Blue | ...

Atuin is an imperative language, consisting of sequences of commands applied to expressions, therefore we define expressions (Exp) and procedures (Turtle) mutually. Expressions can be constants or variables, and combined by infix operators. Additionally, we have code blocks in expressions to pass to higher order procedures.

data Exp = Infix Op Exp Exp
    | Var Id
    | ConstConst
    | Block Turtle

data Op = Plus | Minus | Times | Divide | ...

Procedures define sequences of potentially side-effecting turtle operations. There can be procedure calls, turtle commands, and some simple control structures. Pass defines an empty code block:

data Turtle = Call Id [Exp]
    | TurtleCommand
    | Seq Turtle Turtle
    | If Exp Turtle Turtle
    | Let Id Exp Turtle
    | Eval Exp
    | Pass

The turtle can be moved forward, turned left or right, or given a different pen colour. The pen can also be raised, to allow the turtle to move without drawing.

data Command = Fd Exp | RightT Exp | LeftT Exp
    | Colour Exp | PenUp | PenDown

As with the λ-calculus compiler in Section 3, a complete program consists of a collection of definitions, where definitions include a list of formal parameters and the program definition:

type Proc = ([Id], Turtle)
typeDefs = [(Id, Proc)]

4.2 Compiling

While Atuin is a different kind of language from the λ-calculus, with complicating factors such as a global state (the turtle), imperative features, and dynamic type checking, the process of constructing a compiler follows the same general
Compiling Primitives  The first step in defining a compiler for Atuin is to define primitive operations as Epic functions. The language is dynamically typed, therefore we will need primitive operations to check that they are operating on the correct types. We define functions which construct Epic code for building values, effectively using a single algebraic datatype to capture all possible runtime values (i.e. values are “uni-typed” [15]).

```
mkint  i = con_ 0 @@ i  
mkchar c = con_ 1 @@ c  
mkbool b = con_ 2 @@ b  
mkcol c = con_ 3 @@ c
```

Correspondingly, we can extract the concrete values safely from this structure, checking that the value is the required type:

```
getInt x = case_ x  
  [con 0 \ (x :: Expr) -> x,  
   defaultcase (error_ "Not an Int")]
```

Similarly, `getChar`, `getBool` and `getCol` check and extract values of the appropriate type. Using these, it is simple to define primitive arithmetic operations which check they are operating on the correct type, and report an error if not.

```
primPlus  x y = mkint $ op_ plus_ (getInt x) (getInt y)  
primMinus x y = mkint $ op_ minus_ (getInt x) (getInt y)  
primTimes x y = mkint $ op_ times_ (getInt x) (getInt y)  
primDivide x y = mkint $ op_ divide_ (getInt x) (getInt y)
```

Graphics Operations  We use the Simple DirectMedia Layer\(^4\) (SDL) to implement graphics operations. We implement C functions to interact with SDL, and use Epic’s foreign function interface to call these functions. For example:

```
void* startSDL(int x, int y);  
void  drawLine(void* surf,  
               int x, int y, int ex, int ey,  
               int r, int g, int b, int a);
```

The `startSDL` function opens a window with the given dimensions, and returns a pointer to a `surface`, on which we can draw; `drawLine` draws a line on a surface, between the given locations, and in the given colour, specified as red, green, blue and alpha channels.

We represent colours as a 4-tuple \((r, g, b, a)\). Drawing a line in Epic involves extracting the red, green, blue and alpha components from this tuple, then calling the C `drawLine` function. Recall that to make a foreign function call, we give the

\(^4\) http://libsdl.org/
types of each argument explicitly so that Epic will know how to convert from internal values to C values:

```haskell
drawLine :: Expr -> Expr -> Expr ->
    Expr -> Expr -> Expr -> Term
drawLine surf x y ex ey col
    = case_ (rgba col)
        [tuple (\ r g b a ->
            foreign_ tyUnit "drawLine"
                [(surf, tyPtr),
                (x, tyInt), (y, tyInt), (ex, tyInt), (ey, tyInt),
                (r, tyInt), (g, tyInt), (b, tyInt), (a, tyInt)]) ]
```

When we compile Atuin programs, we treat the turtle state as a tuple \((s, x, y, d, c, p)\) where \(s\) is a pointer to the SDL surface, \(x\) and \(y\) give the turtle’s location, \(d\) gives the turtle’s direction, \(c\) gives the colour and \(p\) gives the pen state (a boolean, false for up and true for down). Note that this state is not accessible by the programmer, so we do not need any dynamic type checking of each component.

To implement the `forward` operation, for example, we take the current turtle state, update the position according to the distance given and the current direction, and if the pen is down, draws a line from the old position to the new position.

```haskell
forward :: Expr -> Expr -> Term
forward st dist = case_ st
    [tuple (\ (surf :: Expr) (x :: Expr) (y :: Expr)
                (dir :: Expr) (col :: Expr) (pen :: Expr) ->
            let_ (op_ plus_ x (op_ times_ (getInt dist) (esin dir)))
            (\x' -> let_ (op_ plus_ y (op_ timesF_ (getInt dist) (ecos dir)))
             (\y' -> if_ pen (fn "drawLine" @@ surf @@ x @@ y
                        @@ x' @@ y' @@ col) unit_ ->
                tuple_ @@ surf @@ x' @@ y' @@ dir @@ col @@ pen)))]
```

Note that there is a distinction between Epic code fragments, and Epic functions. We have used `getInt`, `esin` and `ecos` as inline Haskell functions, using Haskell application, but `drawLine` is applied as a separately defined Epic function, using Epic’s application operator (@@).

### Compiling Programs

Programs return an updated turtle state, and possibly perform some side-effects such as drawing. We can think of Atuin definitions with arguments \(a_1\) to \(a_n\) as being translated to an Epic function with a type of the following form:

\[
f : \text{State} \rightarrow a_1 \rightarrow \ldots \rightarrow a_n \rightarrow \text{State}
\]

To compile a complete program, we add the primitive functions we have defined above (line drawing, turtle movement, etc) to the list of basic Epic definitions, and convert the user defined procedures to Epic.
prims = basic_defs ++
  [EpicFn (name "initSDL") initSDL,
   EpicFn (name "drawLine") drawLine,
   EpicFn (name "forward") forward, ... ]

We will need to convert expressions, commands and full programs into Epic terms, so define a type class to capture all of these. Programs maintain the turtle's state (an Epic Expr), and return a new state, so we pass this state to the compiler.

    class Compile a where
      compile :: Expr -> a -> Term

In general, since we have set up all of the primitive operations as Epic functions, compiling an Atuin program consists of directly translating the abstract syntax to the Epic equivalent, making sure the state is maintained. For example, to compile a call, we build an Epic function call and add the current state as the first argument. Epic takes strings as identifiers, so we use fullId :: Id -> String to convert an Atuin identifier to an Epic identifier.

    compile state (Call i es) = app (fn fullId i) @@ state) es
    where app f [] = f
        app f (e:es) = app (f @@ compile state e) es

Where operations are sequenced, we make sure that the state returned by the first operation is passed to the next:

    compile state (Seq x y) = let_ (compile state x) (\state' -> compile state' y)

Atuin has higher order procedures which accept code blocks as arguments. To compile a code block, we simply create a function which takes the turtle state (that is, the state at the time the block is executed, not the state at the time the block is created). Then when it is time to evaluate the block, we pass in the current state. Epic's effect function ensures that a closure is evaluated, but the result is not updated as evaluating the closure may have side effects which may need to be executed again — consider the repeat function above, for example, where the code block should be evaluated on each iteration.

    compile state (Block t) = lazy_ (\st -> compile st t)
    compile state (Eval e) = effect_ (compile state e @@ state)

The rest of the operations are compiled by a direct mapping to the primitives defined earlier. Finally, the main program sets up an SDL surface, creates an initial turtle state, and passes that state to the user defined main function:
\begin{verbatim}
init_turtle surf = tuple_ @@ surf @@
    int 320 @@ int 240 @@ int 180 @@
    col_white @@ bool True
runMain :: Term
runMain = let_ (fn "initSDL" @@ int 640 @@ int 480)
            (\surface ->
                (fn (fullId (mkId "main")) @@ (init_turtle surface)) ++>
                 flipBuffers surface ++> pressAnyKey)
\end{verbatim}

The full source code for Atuin and its compiler is available from Hackage (http://hackage.haskell.org/package/atuin), which covers the remaining technical details of linking compiled programs with SDL.

5 Related Work

Epic is currently used by Agda and Idris [4], as well as the development version of Epigram [6]. Initial benchmarking [5] shows that the code generated by Epic can be competitive with Java and is not significantly worse than C. Epic uses techniques from other functional language back ends [10, 12, 13] but deliberately exposes its core language as an API to make it as reusable as possible. Although there is likely to always be a trade off between reusability and efficiency, exposing the API will make it easier for other language researchers to build a new compiler quickly. The Lazy Virtual Machine [8] has similar goals but it designed as a lower level target language, rather than a high level API.

6 Conclusion

Epic provides a simple path for language researchers to convert experimental languages (e.g. experimenting with new type systems or domain specific language design) into larger scale, usable tools, by providing an API for generating a compiler, dealing with well-understood but difficult to implement problems such as lambda lifting, code generation, interfacing with foreign functions and garbage collection.

In this paper we have seen two examples of languages which can be compiled via Epic, both functionally based, but with different features. The high-level recipe for each is the same: define primitive functions as run-time support, then translate the abstract syntax into concrete Epic functions, using a combinator style API.

**Future work** Epic is currently used in practice by a dependently typed functional language, Idris [3], and experimentally by Agda [11] and Epigram [6]. Future work will therefore have an emphasis on providing an efficient executable environment for these and related languages. An interesting research question, for example, is whether the rich type systems of these languages can be used to
guide optimisation, and if so how to present the information gained by the type system to the compiler.

Currently, Epic compiles to machine code via C, using the Boehm conservative garbage collector [2]. While this has been reasonably efficient in practice, we believe that an LLVM based implementation [7, 14] with accurate garbage collection would be more appropriate. Of course, any language which uses Epic as a back end will stand to gain from future optimisation efforts!

References

Towards Modular Compilers for Effects

Laurence E. Day and Graham Hutton

Functional Programming Laboratory
School of Computer Science
University of Nottingham, UK

Abstract. Compilers are traditionally factorised into a number of separate phases, such as parsing, type checking, code generation, etc. However, there is another potential factorisation that has received comparatively little attention: the treatment of separate language features, such as mutable state, input/output, exceptions, concurrency and so forth. In this article we focus on the problem of modular compilation, in which the aim is to develop compilers for separate language features independently, which can then be combined as required. We summarise our progress to date, issues that have arisen, and further work.

Keywords: Modularity, Haskell, Compilation, Monads

1 Introduction

The general concept of modularity can be defined as the degree to which the components of a system may be separated and recombined. In the context of computer programming, this amounts to the desire to separate the components of a software system into independent parts whose behaviour is clearly specified, and can be combined in different ways for different applications. Modularity brings many important benefits, including the ability to break down larger problems into smaller problems, to establish the correctness of a system in terms of the correctness of its components, and to develop general purpose components that are reusable in different application domains.

In this article we focus on the problem of implementing programming languages themselves in a modular manner. In their seminal article, Liang, Hudak and Jones showed how to implement programming language interpreters in a modular manner with respect to different language features, using the notion of monad transformers [8]. In contrast, progress in the area of modular compilers has been more limited, and at present there is no standard approach to this problem. In this article we report on our progress to date on the problem of implementing modular compilers. In particular, we show how:

- Modular syntax for a language can be defined using the à la carte approach to extensible data types that was recently developed by Swierstra [13];
- Modular semantics for a language can be defined by combining the à la carte and modular interpreters techniques, following the lead of Jaskelioff [6];
Modular compilers can be viewed as modular interpreters that produce code that corresponds to an operational semantics of the source program; modular machines that execute the resulting code can also be defined using the notion of modular interpreters.

We illustrate our techniques using a simple language with two computational features, in the form of arithmetic and exceptions. The article is aimed at functional programmers with a basic knowledge of interpreters, compilers and monads, but we do not assume specialist knowledge of monad transformers, modular interpreters, or the à la carte technique. The Haskell code from the article is available from the authors’ web pages.

2 Setting The Scene

In this section we set the scene for the rest of the paper by introducing the problem we are trying to solve. In particular, we begin with a small arithmetic language for which we define four components: syntax, semantics, compiler and virtual machine. We then extend the language with a simple effect in the form of exceptions, and observe how these four components must be changed in light of the new effect. As we shall see, such extensions cut across all aspects and require the modification of existing code in each case.

2.1 A Simple Compiler

Consider a simple language Expr comprising integer values and binary addition, for which we can evaluate expressions into an integer:

```haskell
data Expr = Val Value | Add Expr Expr

data Value = Int

eval :: Expr -> Value
eval (Val n) = n
eval (Add x y) = eval x + eval y
```

Evaluation of expressions in this manner corresponds to giving a denotational semantics to the Expr datatype [12]. Alternatively, expressions can be compiled into a sequence of low-level instructions to be operated upon by a virtual machine, the behaviour of which corresponds to a (small-step) operational semantics [2]. We can compile an expression to a list of operations as follows:

```haskell
type Code = [Op]
data Op = PUSH Int | ADD

comp :: Expr -> Code
```

116
comp c = comp' c []

comp' :: Expr -> Code -> Code
comp' (Val n) c = PUSH n : c
comp' (Add x y) c = comp' x (comp' y (ADD : c))

Note that the compiler is defined in terms of an auxiliary function comp' that takes an additional Code argument that plays the role of an accumulator, which avoids the use of append and leads to simpler proofs [3]. We execute the resulting Code on a virtual machine that operates using a Stack:

type Stack = [Item]
data Item = INT Value
exec :: Code -> Stack
exec c = exec' c []
exec' :: Code -> Stack -> Stack
exec' [] s = s
exec' (PUSH n : c) s = exec' c (INT n : s)
exec' (ADD : c) s = let (INT y : INT x : s') = s in
exec' c (INT (x + y) : s')

The correctness of the compiler can now be captured by stating that the result of evaluating an expression is the same as first compiling, then executing, and finally extracting the result value from the top of the Stack (using an auxiliary function extr), which can be expressed in diagrammatic form as follows:

2.2 Adding a New Effect

Suppose now that we wish to extend our language with a new effect, in the form of exceptions. We consider what changes will need to be made to the languages syntax, semantics, compiler and virtual machine as a result of this extension. First of all, we extend the Expr datatype with two new constructors:

data Expr = ... | Throw | Catch Expr Expr

The Throw constructor corresponds to an uncaught exception, while Catch is a handler construct that returns the value of its first argument unless it is an uncaught exception, in which case it will evaluate the second argument.
From a semantic point of view, adding exceptions to the language requires changing the result type of the evaluation function from `Value` to `Maybe Value` in order to accommodate potential failure when evaluating expressions. In turn, we must rewrite the semantics of values and addition accordingly, and define appropriate semantics for throwing and catching.

```haskell
eval :: Expr -> Maybe Value
eval (Val n) = return n
eval (Add x y) = eval x >>= \n ->
  eval y >>= \m ->
  return (n + m)
eval Throw = mzero
eval (Catch x h) = eval x ‘mplus’ eval h
```

In the above code, we exploit the fact that `Maybe` is monadic [11, 14, 7, 15]. In particular, we utilise the basic operations of the `Maybe` monad, namely `return`, which converts a pure value into an impure result, `(>>=)`, used to sequence computations, `mzero`, corresponding to failure, and `mplus`, for sequential choice.

Finally, in order to compile exceptions we must introduce new operations in the virtual machine and extend the compiler accordingly [4]:

```haskell
data Op = ... | THROW | MARK Code | UNMARK

comp :: Expr -> Code
comp e = comp' e []

comp' :: Expr -> Code -> Code
comp' (Val n) c = PUSH n : c
comp' (Add x y) c = comp' x (comp' y (ADD : c))
comp' Throw c = THROW : c
comp' (Catch x h) c = MARK (comp' h c) : comp' x (UNMARK : c)
```

Intuitively, `THROW` is an operation that throws an exception, `MARK` tells the virtual machine to make a record on the stack of the `Code` to be performed should the first argument of a `Catch` fail, and `UNMARK` indicates that no uncaught exceptions were encountered and hence the record of the handler `Code` can be removed. Note that the accumulator plays a key role in the compilation of `Catch`, being used in two places to represent the code to be executed after the current compilation.

Because we now need to keep track of handler code on the stack as well as integer values, we must extend the `Item` datatype and also extend the virtual machine to cope with the new operations and the potential for failure:

```haskell
data Item = ... | HAND Code

exec :: Code -> Maybe Stack
exec c = exec' c []
```
exec' :: Code -> Stack -> Maybe Stack
exec' [] s = return s
exec' (PUSH n : c) s = exec' c (INT n : s)
exec' (ADD : c) s = let (INT y : INT x : s') = s in
  exec' c (INT (x + y) : s')
exec' (THROW : _) s = unwind s
exec' (MARK h : c) s = exec' c (HAND h : s)
exec' (UNMARK : c) s = let (v : HAND _ : s') = s in
  exec' c (v : s')

The auxiliary unwind function implements the process of invoking handler code in the case of a caught exception, by executing the topmost Code record on the execution stack, failing if no such record exists:

unwind :: Stack -> Maybe Stack
unwind [] = mzero
unwind (INT _ : s) = unwind s
unwind (HAND h : s) = exec' h s

2.3 The Problem

As we have seen with the simple example in the previous section, extending the language with a new effect results in a large number of changes to existing code. In particular, we needed to extend three datatypes (Expr, Op and Item), change the return type and existing definition of three functions (eval, exec and exec’), and extend the definition of all the functions involved.

The need to modify and extend existing code for each effect we wish to introduce to our language is clearly at odds with the desire to structure a compiler in a modular manner and raises a number of problems. Most importantly, changing code that has already been designed, implemented, tested and (ideally) proved correct is bad practice from a software engineering point of view [16]. Moreover, the need to change existing code requires access to the source code, and demands familiarity with the workings of all aspects of the language rather than just the feature being added. In the remainder of this paper we will present our work to date on the problem of modular compilation.

3 Modular Effects

In the previous section, we saw one example of the idea that computational effects can be modelled using monads. Each monad normally corresponds to a single effect, and because most languages involve more than one effect, the issue of how to combine monads quickly arises. In this section, we briefly review the approach based upon monad transformers [8].

In Haskell, monad transformers have the following definition:

class MonadTrans t where
  lift :: Monad m => m a -> t m a
Intuitively, a monad transformer is a type constructor $t$ which, when applied to a monad $m$, produces a new monad $t m$. Monad transformers are also required to satisfy a number of laws, but we omit the details here. Associated with every monad transformer is the operation $\text{lift}$, used to convert from values in the base monad $m$ to the new monad $t m$. By way of example, the following table summarises five commonly utilised computational effects, their monad transformer types and the implementations of these types:

<table>
<thead>
<tr>
<th>Effect</th>
<th>Transformer Type</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions</td>
<td>$\text{ErrorT } m \ a$</td>
<td>$m \ (\text{Maybe } a)$</td>
</tr>
<tr>
<td>State</td>
<td>$\text{StateT } s \ m \ a$</td>
<td>$s \rightarrow m \ (a, s)$</td>
</tr>
<tr>
<td>Environment</td>
<td>$\text{ReaderT } r \ m \ a$</td>
<td>$r \rightarrow m \ a$</td>
</tr>
<tr>
<td>Logging</td>
<td>$\text{WriterT } w \ m \ a$</td>
<td>$m \ (a, w)$</td>
</tr>
<tr>
<td>Continuations</td>
<td>$\text{ContT } r \ m \ a$</td>
<td>$(a \rightarrow m \ r) \rightarrow m \ r$</td>
</tr>
</tbody>
</table>

The general strategy is to stratify the required effects by starting with a base monad, often the $\text{Identity}$ monad, and applying the appropriate transformers. There are some constraints regarding the ordering; for example, certain effects can only occur at the innermost level and certain effects do not commute [8], but otherwise effects can be ordered arbitrarily.

To demonstrate the concept of transformers, we will examine the transformer for exceptions in more detail. Its type constructor is declared as follows:

```haskell
newtype ErrorT m a = E { run :: m (Maybe a) }
```

Note that applying the $\text{ErrorT}$ constructor to the $\text{Identity}$ monad is equivalent to using the $\text{Maybe}$ monad directly. It is straightforward to declare $\text{ErrorT}$ as a member of both the $\text{Monad}$ and $\text{MonadTrans}$ classes:

```haskell
instance Monad m => Monad (ErrorT m) where
  return :: a -> ErrorT m a
  return a = E $ return (Just a)

  (>>=) :: ErrorT m a -> (a -> ErrorT m b) -> ErrorT m b
  (E m) >>= f = E $ do v <- m
                         case v of
                           Nothing -> return Nothing
                           Just a -> run (f a)

instance MonadTrans ErrorT where
  lift :: m a -> ErrorT m a
  lift m = E $ m >>= \v -> return (Just v)
```

In addition to the general monadic operations, we would like access to other primitive operations related to the particular effect that we are implementing. In this case, we would like to be able to throw and catch exceptions, and we can specify this by having these operations supported by an error monad class:

```haskell
newtype ErrorT m a = E { run :: m (Maybe a) }
```
class Monad m => ErrorMonad m where
    throw :: m a
    catch :: m a -> m a -> m a

We instantiate ErrorT as a member of this class as follows:

instance Monad m => ErrorMonad (ErrorT m) where
    throw :: ErrorT m a
    throw = E $ return Nothing
    catch :: ErrorT m a -> ErrorT m a -> ErrorT m a
    x 'catch' h = E $ do v <- run x
                    case v of
                        Nothing -> run h
                        Just a   -> return v

We can also declare monad transformers as members of effect classes other than their own. Indeed, this is the primary purpose of the lift operation. For example, we can extend StateT to support exceptions as follows:

instance ErrorMonad m => ErrorMonad (StateT s m) where
    throw :: StateT s m a
    throw = lift . throw
    catch :: StateT s m a -> StateT s m a -> StateT s m a
    x 'catch' h = S $ \s -> run s 'catch' run h s

In this manner, a monad that is constructed from a base monad using a number of transformers comes equipped with the associated operations for all of the constituent effects, with the necessary liftings being handled automatically.

Going back to the earlier remark that some transformers do not commute, the semantics resulting from lifting in this manner need not be unique for a set of transformers. For example, consider a monad supporting both exceptions and state. Depending on the order in which this monad is constructed, we may or may not either access the state after an exception is thrown, as reflected in the types \( s \rightarrow (\text{Maybe } a, s) \) and \( s \rightarrow \text{Maybe } (a, s) \). Semantic differences such as these are not uncommon when combining effects, and reflect the fact that the order in which effects are performed makes an observable difference.

Now that we have reviewed how to handle effects in a modular way, let us see how to modularise the syntax of a language.

4 Modular Syntax and Semantics

We have seen that adding extra constructors to a datatype required the modification of existing code. In this section, we review the modular approach to datatypes and functions over them put forward by Swierstra [13], known as datatypes à la carte, and show how it can be used to obtain modular syntax and semantics for the language Expr previously described.
4.1 Datatypes à La Carte

The underlying structure of an algebraic datatype such as Expr can be captured by a constructor signature. Consider signature functors for the arithmetic and exceptional components of the Expr datatype, defined as follows:

```haskell
data Arith e = Val Int | Add e e

data Except e = Throw | Catch e e
```

These definitions capture the non-recursive aspects of expressions, in the sense that Val and Throw have no subexpressions, whereas Add and Catch have two. We can easily declare Arith and Except as functors in Haskell:

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor Arith where
  fmap :: (a -> b) -> Arith a -> Arith b
  fmap f (Val n) = Val n
  fmap f (Add x y) = Add (f x) (f y)

instance Functor Except where
  fmap :: (a -> b) -> Except a -> Except b
  fmap f Throw = Throw
  fmap f (Catch x h) = Catch (f x) (f h)
```

For any functor f, its induced recursive datatype, Fix f, is defined as the least fixpoint of f. In Haskell, this can be implemented as follows [10]:

```haskell
data Fix f = In (f (Fix f))
```

For example, Fix Arith is the language of integers and addition, while Fix Except is the language comprising throwing and catching exceptions. We shall see later on in this section how these languages can be combined.

Given a functor f, it is convenient to use a fold operator (sometimes called a catamorphism) [9] in order to define functions over Fix f [13]:

```haskell
fold :: Functor f => (f a -> a) -> Fix f -> a
fold f (In t) = f (fmap (fold f) t)
```

The parameter of type f a -> a is called an f-algebra, and can be intuitively viewed as a directive for processing each constructor of a functor. Given such an algebra and a value of type Fix f, the fold operator exploits both the functorial and recursive characteristics of Fix to process recursive values.

The aim now is to take advantage of the above machinery to define a semantics for our expression language in a modular fashion. Such semantics will have type Fix f -> m Value for some functor f and monad m; we could also
abstract over the value type, but for simplicity we do not consider this here. To define functions of this type using fold, we require an appropriate evaluation algebra, which notion we capture by the following class declaration:

\[
\text{class (Monad } m, \text{ Functor } f) \Rightarrow \text{ Eval } f \ m \text{ where}
\]
\[
\text{evalAlg} :: f (m \text{ Value}) \rightarrow m \text{ Value}
\]

Using this notion, it is now straightforward to define algebras that correspond to the semantics for both the arithmetic and exception components:

\[
\text{instance Monad } m \Rightarrow \text{ Eval Arith } m \text{ where}
\]
\[
\text{evalAlg} :: \text{ Arith (m Value)} \rightarrow m \text{ Value}
\]
\[
\text{evalAlg (Val } n) = \text{return } n
\]
\[
\text{evalAlg (Add } x \ y) = x \triangleright\triangleright \ n \rightarrow
\]
\[
y \triangleright\triangleright \ m \rightarrow
\]
\[
\text{return } (n + m)
\]

\[
\text{instance ErrorMonad } m \Rightarrow \text{ Eval Except } m \text{ where}
\]
\[
\text{evalAlg} :: \text{ Except (m Value)} \rightarrow m \text{ Value}
\]
\[
\text{evalAlg (Throw)} = \text{throw}
\]
\[
\text{evalAlg (Catch } x \ h) = x \text{ ‘catch‘ } h
\]

There are three important points to note about the above declarations. First of all, the semantics for arithmetic have now been completely separated from the semantics for exceptions, in particular by way of two separate instance declarations. Secondly, the semantics are parametric in the underlying monad, and can hence be used in many different contexts. And finally, the operations that the underlying monad is required to support are explicitly quantified by class constraints, e.g. in the case of Except the monad must be an ErrorMonad.

With this machinery in place, we can now define a general evaluation function of the desired type by folding an evaluation algebra:

\[
\text{eval} :: (\text{Monad } m, \text{ Eval } f \ m) \Rightarrow \text{ Fix } f \rightarrow m \text{ Value}
\]
\[
\text{eval} = \text{fold evalAlg}
\]

Note that this function is both modular in the syntax of the language and parametric in the underlying monad. However, at this point we are only able to take the fixpoints of Arith or Except, not both. We need a way to combine signature functors, which is naturally done by taking their coproduct (disjoint sum) [8]. In Haskell, the coproduct of two functors can be declared as follows:

\[
\text{data (f :+: g) e} = \text{Inl (f e) | Inr (g e)}
\]

\[
\text{instance (Functor } f, \text{ Functor } g) \Rightarrow \text{ Functor (f :+: g)} \text{ where}
\]
\[
\text{fmap} :: (a \rightarrow b) \rightarrow (f :+: g) a \rightarrow (f :+: g) b
\]
\[
\text{fmap } f (\text{Inl } x) = \text{Inl (fmap } f x)
\]
\[
\text{fmap } g (\text{Inr } y) = \text{Inr (fmap } g y)
\]
It is then straightforward to obtain a coproduct of evaluation algebras:

\[
\text{instance (Eval } f \text{ m, Eval } g \text{ m) } \Rightarrow \text{ Eval (} f :+: g \text{) m where}
\]
\[
\text{evalAlg } :: (f :+: g) \text{ (m Value)} \rightarrow m \text{ Value}
\]
\[
\text{evalAlg (Inl } x \text{)} = \text{ evalAlg } x
\]
\[
\text{evalAlg (Inr } y \text{)} = \text{ evalAlg } y
\]

The general evaluation function defined above can now be used to give a
semantics to languages constructed using multiple features by simply taking the
coproduct of their signature functors. Unfortunately, there are three problems
with this approach. First of all, the need to include fixpoint and coproduct tags
(\text{In}, \text{Inl} and \text{Inr}) in values is cumbersome. Secondly, the extension of an existing
syntax with additional operations may require the modification of existing tags,
which breaks modularity. And finally, \text{Fix (} f :+: g \text{)} and \text{Fix (} g :+: f \text{)} are
isomorphic as languages, but require equivalent values to be tagged in different
ways. The next section reviews how Swierstra resolves these problems [13].

4.2 Smart Constructors

We need a way of automating the injection of values into expressions such that
the appropriate sequences of fixpoint and coproduct tags are prepended. This
can be achieved using the concept of a subtyping relation on functors, which
can be formalised in Haskell by the following class declaration, within which the
function \text{inj} injects a value from a subtype into a supertype:

\[
\text{class (Functor sub, Functor sup) } \Rightarrow \text{ sub } :<: \text{ sup where}
\]
\[
\text{inj } :: \text{ sub } a \rightarrow \text{ sup } a
\]

It is now straightforward to define instance declarations to ensure that \text{f}
is a
subtype of any coproduct containing \text{f}, but we omit the details here. Using the
notion of subtyping, we can define an injection function,

\[
\text{inject } :: (g :<: f) \Rightarrow g \text{ (Fix } f \text{)} \rightarrow \text{ Fix } f
\]

\[
\text{inject } = \text{ In } \cdot \text{ inj}
\]

which then allows us to define smart constructors which bypass the need to tag
values when embedding them in expressions:

\[
\text{val } :: (\text{Arith :<: } f) \Rightarrow \text{ Int } \rightarrow \text{ Fix } f
\]
\[
\text{val } n = \text{ inject } (\text{Val } n)
\]
\[
\text{add } :: (\text{Arith :<: } f) \Rightarrow \text{ Fix } f \rightarrow \text{ Fix } f \rightarrow \text{ Fix } f
\]
\[
\text{add } x \text{ y} = \text{ inject } (\text{Add } x \text{ y})
\]
\[
\text{throw } :: (\text{Except :<: } f) \Rightarrow \text{ Fix } f
\]
\[
\text{throw } = \text{ inject } \text{ Throw}
\]
\[
\text{catch } :: (\text{Except :<: } f) \Rightarrow \text{ Fix } f \rightarrow \text{ Fix } f \rightarrow \text{ Fix } f
\]
\[
\text{catch } x \text{ h} = \text{ inject } (\text{Catch } x \text{ h})
\]
Note the constraints stating that \( f \) must have the appropriate signature functor as a subtype; for example, in the case of \texttt{val}, \( f \) must support arithmetic.

### 4.3 Putting It All Together

We have now achieved our goal of being able to define modular language syntax. Using the smart constructors, we can define values within languages given as fixpoints of coproducts of signature functors. For example:

\[
\begin{align*}
ex1 :: & \text{Fix Arith} \\
& \text{ex1 = val 18 'add' val 24} \\
\end{align*}
\]

\[
\begin{align*}
ex2 :: & \text{Fix Except} \\
& \text{ex2 = throw 'catch' throw} \\
\end{align*}
\]

\[
\begin{align*}
ex3 :: & \text{Fix (Arith :+: Except)} \\
& \text{ex3 = throw 'catch' (val 1337 'catch' throw)} \\
\end{align*}
\]

The types of these expressions can be generalised using the subtyping relation, but for simplicity we have given fixed types above.

In turn, the meaning of such expressions is given by our modular semantics:

\[
\begin{align*}
> \text{eval ex1 :: Value} \\
> 42 \\

> \text{eval ex2 :: Maybe Value} \\
> \text{Nothing} \\

> \text{eval ex3 :: Maybe Value} \\
> \text{Just 1337} \\
\end{align*}
\]

Note the use of explicit type signatures to determine the resulting monad. Whilst we have used \texttt{Identity} (implicitly) and \texttt{Maybe} above, any monad satisfying the required constraints can be used. For example, both \texttt{eval ex1 :: Maybe Value} and \texttt{eval ex2 :: [Value]} are also valid uses of the evaluation function.

### 5 Modular Compilers

With the techniques we have described, we can now construct a modular compiler for our expression language. First of all, we define the \texttt{Code} datatype in a modular manner as the coproduct of signature functors corresponding to the arithmetic and exceptional operations of the virtual machine:

\[
\begin{align*}
type & \text{Code} = \text{Fix (ARITH :+: EXCEPT :+: EMPTY)} \\
\text{data ARITH } e = & \text{PUSH Int e | ADD e} \\
\end{align*}
\]
data EXCEPT e = THROW e | MARK Code e | UNMARK e

data EMPTY e = NULL

There are two points to note about the above definitions. First of all, rather than defining the \( \text{Op} \) type as a fixpoint (where \( \text{Code} \) is a list of operations), we have combined the two types into a single type defined using \( \text{Fix} \) in order to allow code to be processed using \( \text{fold} \); note that \( \text{EMPTY} \) now plays the role of the empty list. Secondly, the first argument to \( \text{MARK} \) has explicit type \( \text{Code} \) rather than general type \( e \), which is undesirable as this goes against the idea of treating code in a modular manner. However, this simplifies the definition of the virtual machine and we will return to this point in the conclusion.

The desired type for our compiler is \( \text{Fix } f \rightarrow (\text{Code} \rightarrow \text{Code}) \) for some signature functor \( f \) characterising the syntax of the source language. To define such a compiler using the general purpose \( \text{fold} \) operator, we require an appropriate \textit{compilation algebra}, which notion we define as follows:

\[
\begin{align*}
\text{class Functor } f & \Rightarrow \text{Comp } f \text{ where} \\
\text{compAlg} & \colon f (\text{Code} \rightarrow \text{Code}) \rightarrow (\text{Code} \rightarrow \text{Code})
\end{align*}
\]

In contrast with evaluation algebras, no underlying monads are utilised in the above definition, because the compilation process itself does not involve the manifestation of effects. We can now define algebras for both the arithmetic and exceptional aspects of the compiler in the following manner:

\[
\begin{align*}
\text{instance Comp Arith where} & \\
\text{compAlg} & \colon \text{Arith} (\text{Code} \rightarrow \text{Code}) \rightarrow (\text{Code} \rightarrow \text{Code}) \\
\text{compAlg} (\text{Val } n) & = \text{pushc } n \\
\text{compAlg} (\text{Add } x \ y) & = x \ . \ y \ . \ \text{addc}
\end{align*}
\]

\[
\begin{align*}
\text{instance Comp Except where} & \\
\text{compAlg} & \colon \text{Except} (\text{Code} \rightarrow \text{Code}) \rightarrow (\text{Code} \rightarrow \text{Code}) \\
\text{compAlg} \text{ Throw} & = \text{throwc} \\
\text{compAlg} (\text{Catch } x \ h) & = \ \lambda c \rightarrow h \ c \ \text{‘markc’ } x \ (\text{unmarkc } c)
\end{align*}
\]

In a similar manner to the evaluation algebras defined in §4.1, note that these definitions are modular in the sense that the two language features are being treated completely separately from each other. We also observe that because the carrier of the algebra is a function, the notion of appending code in the \text{Add} case corresponds to function composition. Finally, the smart constructors \text{pushc}, \text{addc}, etc are defined in the obvious manner:

\[
\begin{align*}
\text{pushc} & \colon \text{Int} \rightarrow \text{Code} \rightarrow \text{Code} \\
\text{pushc } n \ c & = \text{inject} \ (\text{PUSH } n \ c)
\end{align*}
\]

\[
\begin{align*}
\text{addc} & \colon \text{Code} \rightarrow \text{Code} \\
\text{addc } c & = \text{inject} \ (\text{ADD } c)
\end{align*}
\]
The other smart constructors are defined similarly. It is now straightforward to define a general compilation function of the desired type by folding a compilation algebra, supplied with an initial accumulator `empty`:

\[
\begin{align*}
\text{comp} & : \text{Comp f} \Rightarrow \text{Fix f} \rightarrow \text{Code} \\
\text{comp e} & = \text{comp}' \ e \ \text{empty} \\
\text{comp}' & : \text{Comp f} \Rightarrow \text{Fix f} \rightarrow (\text{Code} \rightarrow \text{Code}) \\
\text{comp}' \ e & = \text{fold} \ \text{compAlg} \ e \\
\text{empty} & : \text{Code} \\
\text{empty} & = \text{inject} \ \text{NULL}
\end{align*}
\]

6 Towards Modular Machines

The final component of the modularisation of our small example language is to define the virtual machine. Defining the underlying `Stack` datatype in a modular manner is now straightforward:

\[
\begin{align*}
\text{type Stack} & = \text{Fix} \ (\text{Integer :+: Handler :+: EMPTY}) \\
\text{data Integer e} & = \text{VAL} \ \text{Int} \\
\text{data Handler e} & = \text{HAND} \ \text{Code}
\end{align*}
\]

As we saw in §2.1, the virtual machine for arithmetic had type `Code \rightarrow Stack \rightarrow Stack`, while in §2.2, the extension to exceptions required modifying the type to `Code \rightarrow Stack \rightarrow \text{Maybe Stack}`. Generalising from these examples, we seek to define a modular execution function of type `Code \rightarrow Stack \rightarrow m Stack` for an arbitrary monad `m`. We observe that `Stack \rightarrow m Stack` is a state transformer, and define the following abbreviation:

\[
\begin{align*}
\text{type StackTrans m} \ a & = \text{StateT} \ \text{Stack} \ m \ a
\end{align*}
\]

Using this abbreviation, we now seek to define a general purpose execution function of type `Fix f \rightarrow StackTrans m (\_)` for some signature functor `f` characterising the syntax of the code, and where `_` represents a void result type. In a similar manner to evaluation and compilation algebras that we introduced previously, this leads to the following notion of an execution algebra,

\[
\begin{align*}
\text{class} \ (\text{Monad m}, \ \text{Functor f}) & \Rightarrow \text{Exec} \ f \ m \ \text{where} \\
\text{execAlg} & : f \ (\text{StackTrans} \ m \ ()) \rightarrow \text{StackTrans} \ m \ ()
\end{align*}
\]

for which we define the following three instances:
instance Monad m => Exec ARITH m where
  execAlg :: ARITH (StackTrans m ()) -> StackTrans m ()
  execAlg (PUSH n st) = pushs n >> st
  execAlg (ADD st) = adds >> st

instance ErrorMonad m => Exec EXCEPT m where
  execAlg :: EXCEPT (StackTrans m ()) -> StackTrans m ()
  execAlg (THROW _) = unwinds
  execAlg (MARK h st) = marks h >> st
  execAlg (UNMARK st) = unmarks >> st

instance Monad m => Exec EMPTY m where
  execAlg :: EMPTY (StackTrans m ()) -> StackTrans m ()
  execAlg (Null) = stop

The intention is that \texttt{pushs}, \texttt{adds}, etc are the implementations of the semantics for the corresponding operations of the machine, and >> is the standard monadic operation that sequences two effectful computations and ignores their result values (which in this case are void). We have preliminary implementations of each of these operations but these appear more complex than necessary, and we are in the process of trying to define these in a more elegant, structured manner.

Folding an execution algebra produces the general execution function:

\[
\text{exec} :: (\text{Monad } m, \text{Exec } f m) \Rightarrow \text{Fix } f \rightarrow \text{StackTrans } m ()
\]

\[
\text{exec} = \text{fold } \text{execAlg}
\]

Abstracting away from specific computational effects, the correctness of our modular compiler can now be captured by the following diagram:

\[
\begin{array}{ccc}
\text{Expr} & \xrightarrow{\text{eval}} & m \text{ Value} \\
\downarrow^{\text{comp}} & & \downarrow^{m \text{ extr}} \\
\text{Code} & \xrightarrow{\text{exec}} & m \text{ Stack}
\end{array}
\]

7 Summary and Conclusion

In this article we reported on our work to date on the problem of implementing compilers in a modular manner with respect to different computational effects that may be supported by the source language. In particular, we showed how modular syntax and semantics for a simple source language can be achieved by combining the à la carte approach to extensible datatypes with the monad transformers approach to modular interpreters, and outlined how a modular compiler and virtual machine can be achieved using the same technology.

However, this is by no means the end of the story, and much remains to be done. Topics for further work include how to support a more modular code
Towards Modular Compilers for Effects - Day and Hutton

type when compiling exceptions, simplifying the implementation of the virtual machine operations, considering other effects and languages with binding constructs, formalising the idea that some effects may be ‘compiled away’ and hence do not appear in the virtual machine, utilising the more principled approach to lifting monadic operations developed by Jaskelioff [5], exploring the idea of modular correctness proofs, and the relationship to Harrison’s work [1].

Acknowledgements

We would like to thank Mauro Jaskelioff and the participants of BCTCS 2011 in Birmingham for useful comments and suggestions.

References

Compiling Hume down to gates

J. Sérot\(^1\) and G. Michaelson\(^2\)

\(^1\) LASMEA, Université Blaise Pascal/CNRS, France,
Jocelyn.Serot@lasmea.univ-bpclermont.fr
\(^2\) Heriot-Watt University, Edinburgh, Scotland
G.Michaelson@hw.ac.uk

Abstract. We describe the implementation of a subset of the Hume programming language on a FPGA architecture at the gate level. Hume is a domain specific language for developing multi-process systems requiring strong static guarantees that resource bounds are met. The compiler produces RT-level, synthesizable VHDL code that can be processed by a standard tool chain to program FPGAs at the gate level. Preliminary results suggest that this compilation route offers substantial opportunities for exploiting fine-grain parallelism in Hume programs. The approach also offers a significantly higher abstraction level than that offered by traditional hardware description languages such as VHDL or Verilog.

1 Introduction

1.1 Overview

Field Programmable Gate Areas (FPGAs) have long been promoted as a solution to the direct realisation of software in hardware. While CPUs offer very fast execution in hardware of low level instructions, a specific CPU design may not be optimal for individual programs. In contrast, in principle, an FPGA may be directly configured to realise a specific program.

And FPGAs have long been promoted as a way out of the restrictions of specific CPU designs on system scalability. While fabrication technology is rapidly increasing the number of processing elements in multi-core CPUs, nonetheless such cores are necessarily in some fixed configuration which may not be optimal for an arbitrary problem. In contrast, in principle, an FPGA of sufficient size may implement an arbitrary number of processing elements with arbitrary interconnections.

Nonetheless, there are immense practical problems in realising the full potential of FPGAs. In particular, FPGAs are very low level devices requiring expert understanding of hardware concerns to gain best performance. Thus, there has been considerable research into developing both languages for describing FPGA configurations at considerably higher levels of abstraction, and tool chains for seamlessly realising such abstracted configurations in hardware.

130
1.2 High-level imperative FPGA programming

High level hardware description languages (HDLs) are now very widely used for programming FPGAs. Such HDLs typically expose very low level hardware concerns within higher level type and control constructs, and have tool chains to generate gate-level descriptions from register-transfer level descriptions (RTL), for realisation ultimately in an Application Specific Integrated Circuit (ASIC) or configured FPGA.

The widely used Verilog[10] was strongly influenced by C. Similarly, the widely used and somewhat higher level VHDL was strongly influenced by Ada. Both languages offer tool chains to RTL but have very restricted data types and control constructs.

At a higher level again, SystemC is based on library extensions to C++ and Handel-C[4] augments a C subset with hardware-oriented constructs. Both SystemC and Handel-C may be compiled to lower level HDLs for hardware realisation.

Microsoft’s Kiwi project[9] is exploring the use of high level languages for hardware description within the .Net framework. The main focus is on C# with translation to VHDL.

HDLs are substantial improvements on traditional low level diagrammatic hardware design techniques. However, like system programming languages, they offer uneasy compromises between the higher and lower levels. First of all, HDLs usually build from bit or wire level components requiring expert understanding for effective use. Furthermore, they typically offer a subset of a full high level language, for example restricting types and control constructs, while nonetheless retaining high level constructs which may be simulated but not realised in hardware.

Finally, the marriage of a conventional imperative language with hardware oriented extensions is not always amicable. As the anonymous “bob” comments on the Kiwi project[9]:

bob 14 Nov 2010 9:49 PM: the vhdl code was cleaner and easier to understand than the C# code...

1.3 Functional FPGA programming

There are also a number of functional approaches to FPGA programming, drawing on the classic FP strength of higher order abstraction to compose components for hardware realisation.

Lava[3] augments Haskell with modules for hardware description. The Lava tool chain generates VHDL. Sheeran[8] provides a useful reflection on Lava’s origins. Several groups are actively developing Lava, for example Kansas Lava[5].

The Kiwi project has recently been complemented with the use of F#[9], a Standard ML derivation. Here, common middleware for all .Net compliant languages eases the route to VHDL.

All these functional approaches share the same problems of high level imperative languages for FPGA programming, that is the requirement for deep understanding of hardware architectures and restrictions on which high level constructs can be realised in hardware. Here, the inability to realise arbitrary recursion is particularly problematic given the statelessness of functional languages. Furthermore, functional languages bring the long standing difficulty of lack of familiarity for mainstream programmers and hardware designers.

1.4 Hume and FPGAs

Hume[6] is a contemporary language for developing multi-process systems requiring strong static guarantees that resource bounds are met. With roots in polymorphic functional languages, Hume is distinguished by an explicit separation of coordination and expression layer. The coordination layer, for configuring independent communicating processes, is based on concurrent finite state boxes connected by single-buffered wires. The expression layer defines control within boxes and is based on pattern matching on input values to enable general recursive actions to generate output values.

A crucial feature of Hume is that the expression layer is state free, with all local variable instantiations lost between execution cycles. However, in the coordination layer state is retained on wires. In particular, feedback wires from a box’s outputs to its inputs enable individual boxes to retain state between execution cycles, and are the basis of box iteration.

Hume’s tool chain is strongly based on the Hume Abstract Machine (HAM) which provides a unitary locus for consistent implementation and resource analysis. Thus, a standard compiler generates HAM code from Hume which may be:

- interpreted directly on the HAM;
- further compiled to native code, for example via C;
- analysed to identify resource bounds, for example via an amortised type system implemented within Isabelle.

A number of routes have been explored for implementing Hume on FPGAs. These include executing:

- HAM code on the HAM on single Power PC core[1];
- HAM code compiled to native code on single Power PC and microBlaze cores[1];
- Hume compiled directly via C to native code on multiple microBlaze cores[12].

All routes offer consistent, scalable speedup, but are nonetheless markedly slow compared with the equivalent routes on proprietary CPUs.

The Reduceron[7] is a soft core for an FPGA, implementing an abstract graph reduction machine for a minimal non-strict, higher order functional programming
language. It would be interesting to evaluate the Reduceron against the Hume abstract machine on FPGA: it is likely that the performance limitations of lazy evaluation are substantially outweighed by direct implementation of the abstract machine rather than via another soft core.

Nonetheless, we think that Hume’s explicit separation of coordination and control addresses the problems encountered in other high level approaches to FPGA programming as discussed above, and offers a basis for direct realisation via a HDL.

Hume was explicitly intended for use as a multi-level language sharing a common coordination form. Each level reflects different restrictions on expressivity, in particular in the allowed use of types and functional forms, from Hardware Hume (HW-Hume), restricted to pattern matching on bit patterns, to full Hume which is Turing complete. Each level has different formal properties, so HW-Hume has decidable time and space behaviour and full Hume shares all the undecidability restrictions of Turing completeness.

Thus, given a base FPGA realisation of box coordination alone, then the expressivity at the control level might also be varied to reflect the sophistication of hardware compilation. We anticipate that Finite State Hume (FSM-Hume), which augments HW-Hume with fixed size types and arithmetic/logic operations, will be an excellent starting point for direct HDL implementation. Subsequently, Template Hume, which provides a fixed repertoire of higher order functions, offers a framework for exploring functional abstraction in composing hardware components, drawing on the experiences of the pure functional approaches discussed above.

Furthermore, we think that Hume coordination offers an appropriate degree of abstraction from hardware realisation, enabling efficient implementation without requiring deep knowledge of underlying hardware.

Finally, Hume encourages a box-based approach to software design. This, along with the notion of state machines, is very familiar to hardware designers which may ease acceptability beyond our perfectly formed but nonetheless very small community.

In subsequent sections we: introduce Hume and a running example in slightly more detail; give a short presentation of the VHDL programming language; describe our compilation route from Hume to VHDL; explore compilation of the example; and evaluate its performance on an FPGA.

2 mHume

mHume[12] is an evolving experimental version of Hume. Based around the full coordination layer, it provides a platform for exploring the direct compilation of different expression layer instantiations without going through the HAM. The mHume syntax is summarised in Figure 1. [12] describes direct compilation to C. The mHume syntax is summarised in Figure 1.
The example in Figures 2 and 3, from [12], finds the squares of successive integers by repeated addition. Here a minimal expression layer for integers and integer operations is used.

```
1 type integer = int 64;
2 box inc
3   in (n::integer)
4   out (r::integer,n'::integer)
5   match (n) -> (n,n+1);
6 box square
7   in (i::integer,s::integer,
8       c::integer,v::integer)
9   out (o'::integer,s'::integer,
10      c'::integer,v'::integer)
11 match
12   (*,s,0,v) -> (s,*,*,*) |
13   (*,s,c,v) -> (*,s+v,c-1,v) |
14   (i,*,*,*) -> (*,0,i,i);
15 stream output to "std_out";
16 wire inc (inc.n' initially 0)
17    (square.i,inc.n);
18 wire square
19   (inc.r,square.s',square.c',square.v')
20 (output,square.s,square.c,square.v);
```

Fig. 1. mHume syntax.

Fig. 2. Square program.

Fig. 3. Square program Code
Line 1 introduces `integer` as an alias for `int 64`, that is a 64-bit integer. Lines 2 to 5 define a box `inc` (2) with integer input wire `n` (3) and integer output wires `r` and `n'` (4). In line 5, an input is matched with variable `n` to output the value of `n` on wire `r` and `n+1` on wire `n'`. As we shall see, `n` is wired to `n'`. Essentially, `r` is the current and `n` is the next value for squaring Lines 6 to 14 define a box `square` (6) with integer inputs `i`, `s`, `c` and `v` (7 and 8), and integer outputs `o`, `s'`, `c'` and `v'` (9 and 10). In line 12, regardless of the input on `i (*)`, if `c` is 0 then the (final) value from `s` is output on `o`. In line 13, regardless of the value on `i`, `v` is added to `s`, `c` is decremented. In line 14, with a new initial value for `i`, `s` is initialised to 0, and `c` and `v` are initialised to `i`. As we shall see, `s` is wired to `s'`, `c` to `c'` and `v` to `v'`. Essentially, `i` is the value to be squared, `s` is the partial square, `c` counts how often `i` has been added to `s`, and `v` retains the initial value from `i` for repeated addition to `s`. Line 15 associates stream output with standard output. Lines 16 and 17 wire `inc`’s `n` to it’s `n`, and `r` to `square`’s `i`. Finally, lines 18 to 20 wire `square`’s `i` to `inc`’s `r`, `s` to `s'`, `c` to `c'`, `v` to `v'` and `o` to output.

3 VHDL in a nutshell

In this section we introduce the basic concepts of VHDL, focusing on the features which are essential for explaining the principles upon which our compiler is built.

In VHDL, designs are described using a number of modules. Each module consists of a entity and at least one architecture. An entity describes the interface of the module: names and types of the input and output ports in particular. An architecture describes the implementation of the module: how it works. There are basically three ways to write an architecture: structurally (by instantiating other modules and connect them using signals), concurrently (by using a set of concurrent assignments that are re-executed every time signals they depend on change) and sequentially (by encapsulating sequential code within a process). Consider for example, the design depicted in Fig. 4, composed of two simple and gates. The corresponding entity and structural descriptions are given in Fig 5.

![Fig. 4. VHDL example](image1)

```vhdl
ENTITY example IS
    PORT (a,b,c: IN BIT; d: OUT BIT);
END example

ARCHITECTURE structural OF example IS
    SIGNAL s : BIT;
BEGIN
    U1 : entity and_g(beh) PORT MAP (a,b,s);
    U2 : entity and_g(beh) PORT MAP (s,c,d);
END ARCHITECTURE
```

![Fig. 5. VHDL structural description of the example](image2)

```vhdl
ENTITY example IS
    PORT (a,b,c: IN BIT; d: OUT BIT);
END example

ARCHITECTURE structural OF example IS
    SIGNAL s : BIT;
BEGIN
    U1 : entity and_g(beh) PORT MAP (a,b,s);
    U2 : entity and_g(beh) PORT MAP (s,c,d);
END ARCHITECTURE
```
Two modules, named U1 and U2 are instantiated and explicitly connected using signal s. In VHDL, signals correspond to physical wires. Now, the and_g modules can be described by the following entity and behavioral description:

```
ARCHITECTURE beh OF and_g IS
BEGIN
  ENTITY and_g IS
    PORT (i1, i2 : IN BIT;
          o : OUT BIT);
  END example

  PROCESS (i1,i2)
  BEGIN
    o <= i1 and i2;
  END PROCESS
END ARCHITECTURE
```

The behavior of the module is here specified as a (sequential) process. The sensitivity list of this process contains the two inputs, i1 and i2. This means that the process will be executed each time the signal connected to one of those inputs changes. Upon execution, the value (i1 and i2) will be computed and the signal connected to the output o will be updated with this value.

VHDL processes can also make use of variables. In contrast to signals, which are updated concurrently and globally at the end of the execution cycle, a variable is updated as soon as the sequential statement affecting it is executed. They therefore generally do not correspond to physical wires but to registers. Here’s a possible description in VHDL of a 4-bit synchronous counter:

```
ARCHITECTURE beh OF counter IS
  SIGNAL v: UNSIGNED(3 downto 0);
BEGIN
  ENTITY counter IS
    port (
      val: OUT UNSIGNED(3 downto 0);
      clk: STD_LOGIC
    );
  END counter;

  PROCESS (clk)
  BEGIN
    IF rising_edge (clk) THEN
      v := v + 1;
    END IF;
  END PROCESS;

  val <= v;
end beh;
```

Whenever a rising edge occurs on input clk, the internal variable v in incremented. A concurrent statement val <= v takes care of updating the counter output accordingly.

**Synthesis.** Synthesis is the process where a VHDL program is compiled and mapped into an implementation technology such as an FPGA or an ASIC. Not all constructs in VHDL are suitable for synthesis. While different synthesis tools have different capabilities, there exists a common synthetizable subset of VHDL that defines what language constructs and idioms map into common hardware for many synthesis tools. In the current state of the art, programs written at the register transfer level are synthetizable. Register transfer level (RTL) is a level of abstraction in which the circuit’s behavior is defined in terms of data transfers between synchronous registers, all synchronized by the same clock, and the logical operations performed on those data.
4 Compiling Hume to VHDL

Compiling a Hume program to a RT-level VHDL description involves three phases: network generation, box translation, and VHDL transcription.

4.1 Network generation

In this phase, we derive a structural description of the program as a network of components, where a component represent either a box or a wire of the original program. The process is sketched on Fig. 6. The key issue here is that Hume wires are not mapped to physical wires (VHDL signals) but to a dedicated component that we call a buffer. A buffer has one input and one output corresponding to the initial wire and four extra control signals: full, empty, rd and wr. The full (resp. empty) signal tells whether the buffer is ready for reading (resp. writing); it will be used by the box connected to its output (resp. input). The rd (resp. wr) signal, when asserted to 1, actually pops (resp. pushes) the value from (resp. to) the buffer, passing it from the full (resp. empty) to the empty (resp. full) state.

4.2 Box translation

In this phase, each box of the original Hume program is translated into a finite state machine (FSM). This translation process closely follows the dynamic semantics of the language, in which a box can be in two different states: Ready (awaiting input) or BlockedOut (output pending).

Since we are targeting a RT-level description, all transitions will be triggered by a global clock signal. This means that all boxes will actually change state simultaneously. This dramatically simplifies the scheduling algorithm, which can be rewritten as follows:

At each clock cycle
   For each box $b$, in parallel, do
      if $b.state = \text{Ready}$ then

\footnote{Often, and as pointed out by G. Berry in [2] for instance, complex software solutions become trivial when described in hardware, because parallelism comes for free at this level.}
if a fireable rule \( r \) can be found in \( b.rules \) then
   read inputs for rule \( r \);
   \( b.state \) <- \( \text{BlockedOut} \)
end if
else if \( b.state = \text{BlockedOut} \) then
   if outputs for the selected rule \( r \) are writable then
      write outputs for rule \( r \);
      \( b.state \) <- \( \text{Ready} \)
   end if
end if
end for

Each box can be therefore be described as a finite state machine (FSM) having \( nrules + 1 \) states : one state corresponding to the \( \text{Ready} \) state in the previous algorithm and one state per rule, corresponding to the \( \text{BlockedOut} \) state for the corresponding rule. This transformation is illustrated on Fig. 7.

Each transition in the resulting FSM is labeled with a set of conditions and a set of actions (denoted \( \text{Conditions/Actions} \) on the diagram).

At each rule \( r_i \) we associate two sets of conditions and two sets of actions:

- the set \( C_r(r_i) \) denotes the firing conditions for rule \( r_i \), i.e. the conditions on the inputs that must be verified for the corresponding rule to be selected;
- the set \( A_r(r_i) \) denotes the firing actions for rule \( r_i \), i.e. the read operations that must be performed on the inputs when the corresponding rule is selected;
- the set \( C_w(r_i) \) denotes the writing conditions for rule \( r_i \), i.e. the conditions on the outputs that must be verified when the corresponding rule has been selected;
- the set \( A_w(r_i) \) denotes the writing actions for rule \( r_i \), i.e. the write operations that must be performed on the outputs when the corresponding rule has been selected.
There are

- two possible firing conditions: \textit{Avail}(j), meaning that the \textit{j}th input is ready for reading, and \textit{Match}(j, \textit{pat}), meaning that the \textit{j}th input matches pattern \textit{pat};
- one firing action, \textit{Bind}(j, \textit{pat}), meaning "read \textit{j}th input and match the corresponding pattern against pattern \textit{pat}";
- one writing condition, \textit{Avail}(j) meaning that the \textit{j}th output is ready for writing;
- one writing action, \textit{Write}(j, \textit{exp}), meaning "evaluate expression \textit{exp} and write the corresponding value on \textit{j}th.

Table 1 summarizes the rules for computing the sets \(C_r, A_r\) (resp. \(C_w\) and \(A_w\)) from the patterns (resp. expressions) composing a box rule. The FSM obtained for the \textit{square} box introduced in Sec. 2 is given in Fig. 8.

\[
\begin{align*}
C_r[i, \text{var}] &= \{\text{Avail}(i)\} & C_r[i, *] &= \emptyset \\
A_r[i, \text{var}] &= \bigcup_{j=1}^{n} C_r[i, \text{pat}_j] & A_r[i, *] &= \emptyset \\
C_w[\text{exp}_1, ..., \text{exp}_n] &= \bigcup_{j=1}^{n} C_w[i, \text{exp}_j] & A_w[\text{exp}_1, ..., \text{exp}_n] &= \bigcup_{j=1}^{n} A_w[i, \text{exp}_j] \\
C_w[i, \text{exp}] &= \{\text{Avail}(i)\} & A_w[i, *] &= \emptyset \\
A_w[i, *] &= \emptyset
\end{align*}
\]

**Table 1.** Rules for computing the sets \(C_r, A_r, C_w\) and \(A_w\).

### 4.3 Transcription to VHDL

The transcription in VHDL of the network derived in Sec. 4.1 boils down to instantiating the components forming this network and declaring the interconnection wires. The complete Hume program is turned into a VHDL component. The inputs and outputs of this component correspond to the I/O streams declared in this program. This makes it possible to automatically generate a test-bench for the resulting VHDL design, in which the original input (resp. output) data streams are provided (resp. displayed) by specific VHDL processes reading samples from (resp. writing results to) to files for example.

Converting the FSM representation of boxes into VHDL is a little bit more involved. The \textit{Avail} condition on an input (resp. output) is reflected directly into the value of the \textit{full} (resp. \textit{empty}) signal connected to this input (resp. output). But, because reading / writing is actually triggered by asserting the corresponding \textit{rd} (resp \textit{wr}) signals, an extra state must be added for each rule. This transformation is illustrated in Fig. 9 on a simple, mono-rule, example.

\footnote{This evaluation takes place in an environment augmented with the bindings resulting from the corresponding firing action; for the sake of readability environments have been left implicit here. A fully formalized account will be given in the final paper.}
Since the syntax of the box-level expressions is very simple in mHume, the conversion of these expressions can be handled using a very simple syntax-directed function.

Listing 1.1 gives the VHDL code generated for the inc box of the above example. Lines 1-15 give the interface of the component. Hume integers are translated to VHDL std_logic_vectors. As explained on Fig. 6, the n, n_empty and n_rd correspond to the n original input. Similarly, the r, r_full and r_wr (resp. nn, nn_full and nn_wr) correspond to the r (resp. n') original output. The two other input signals are the global clock and a reset initial for hardware initialization. The behavior of the box is explicit in its architecture, lines 17-53. This architecture describes a synchronous FSM. The state variable is declared in line 19, its type being declared in line 18. Here the box has only one rule, so there are three states. The behavior itself is explicit as a process sensitive to the clock and reset signals (line 21). This process uses a internal variable r1_n. This variable memorizes the value obtained when the pattern of rule r1 is bound
The core of the process – which, according to VHDL execution model, is executed whenever the signal clock or reset changes value – is between line 23 and 52. Lines 24-28 handle asynchronous reset: the process state is reset to Ready and read/write signals are set to 0. Lines 30-50 describe what happens when a rising edge occurs on the clock input signal. This part is written in a very classical style, as a big case construct inspecting the value of the process state and, for each possible state, deciding on the actions to perform and the next state. For example, lines 32-35 tell that if process (box) is in the Ready state and a value is available on input n (line 32), then this value is copied (line 33), the read signal is asserted (line 34) and the next state will be R1a (line 35). In state R1a (lines 38-44), the read signal is reset to 0 and the availability of the output link is tested (line 39). If yes, the outputs are written (line 40-43) and the next state will be R1b.

5 Experimental results

Evaluation of the generated VHDL code has been carried out using the Altera Quartus II v9.0 tool chain.

For simulation, two specific, hand-written VHDL processes allow stream inputs and outputs to be read from and written to files.

Simulation results, for the square example introduced in Sec. 2 are displayed on Fig. 10. At this level, the clock period has been arbitrarily fixed to 10 ns. A close inspection of the chronograms shows that it takes four clock cycles to the square box to make an "iteration" (i.e. to increment the sum s and decrement the counter c by 1). Hence, computation of $n^2$ by the box will take $n \times 4$ clock cycles. The optimal version of such an operator, hand-crafted by a trained VHDL programmer, takes $n$ cycles. The four to one ratio is indeed an very acceptable price to pay for abstraction here.

![Fig. 10. Simulation results for the squares example](image)

We performed the synthesis of this example on a Stratix EP1S80B956C6 FPGA. The default parameters for the synthetizer were used. Some views of

5 Currently, a variable is introduced for each pattern appearing in each rule LHS. This can lead to redundancy and will be optimized in future versions of the compiler.
the result are shown in Fig. 13 and Fig. 14. Fig. 13 is a top-level view of the synthetized network. Box labeled box_wire correspond to buffers. The two other boxes implement the square and inc original boxes. Fig. 14 shows the hardware architecture inferred by the synthetizer for the inc box. We can recognize a collection of registers\(^6\) on the left, memorizing the internal outputs (r and nn) of the box and the adder performing the n+1 operation (drawn as a small circle). The rectangular box at center left implements the FSM control.

For performance evaluation, we used a slightly modified application, depicted in Fig. 11. The first box, gen, generates an integer and passes it to the boxes square\(_1\), \ldots, square\(_N\). All these boxes are similar to the square box described above: they compute the square of the integer received by using iterative sum and count operations. The col box simply collects all the results.

Table 12 gives the occupation (number of logic cells used in the chip) and maximum clock frequency for different values of \(N\), the number of square boxes running in parallel. The reported numbers show it should be possible to pack approximately 400 applications in a single FPGA of this category. This represent a vast improvement compared to the previous hardware implementation of mHUme on a FPGA [12] for which the number of parallel boxes was limited by the number of soft-core processors that could be instantiated on a chip (typically less than a dozen). This large amount of parallelism also largely compensate for the relatively small clock frequency, compared to a classical CPU. It takes 40000 cycles, i.e. approximately 270 \(\mu s\) at 150 MHz, on our FPGA to compute the square of 400 integers in the range 1\ldots10000. Reaching the same throughput on a sequential CPU would require that this processor computes the square of an integer in less than 0.7 \(\mu s\), that is, that it performs one sum-count iteration in less than 0.07\(ns\)!

### Table 12. Synthesis results

<table>
<thead>
<tr>
<th>(N)</th>
<th>Occ (#LCs)</th>
<th>Occ (%)</th>
<th>Fmax (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>693</td>
<td>&lt;1</td>
<td>189</td>
</tr>
<tr>
<td>2</td>
<td>885</td>
<td>1</td>
<td>178</td>
</tr>
<tr>
<td>4</td>
<td>1263</td>
<td>2</td>
<td>178</td>
</tr>
<tr>
<td>8</td>
<td>2027</td>
<td>3</td>
<td>165</td>
</tr>
<tr>
<td>16</td>
<td>3550</td>
<td>4</td>
<td>164</td>
</tr>
<tr>
<td>32</td>
<td>6802</td>
<td>8</td>
<td>164</td>
</tr>
<tr>
<td>64</td>
<td>12702</td>
<td>16</td>
<td>139</td>
</tr>
</tbody>
</table>

\(^6\) For the sake of readability, the integer size has here been reduced to 8.

## 6 Conclusion

We have described a simple model and compilation route for implementing a subset of the Hume programming language on a FPGA directly at the gate level using state-of-the art synthesis technology. Preliminary results suggest that this
Fig. 13. Top-level synthetized network for the squares examples

Fig. 14. Synthetized RTL architecture for the inc box
approach route offers vast opportunities for exploiting fine-grain parallelism in Hume programs and a significantly higher abstraction level than that offered by traditional hardware description languages such as VHDL or Verilog.

The Hume subset currently supported is still limited, with data types and computation constructs restricted to integers and simple operations on these integers but it includes all the key features of Hume’s coordination layer. So we are confident that significantly larger and more complex programs than that illustrated in this paper can be implemented on medium-sized FPGAs.

Among the many opportunities for further work, our two priorities are the extension of the expression language supported (integrating conditionals, let expressions, etc.), support for global and external function declarations and optimisations of the generated VHDL code. It should be possible, in particular, to replace looping wires – connecting a box’s output to one of its input – by internal process variables, and therefore minimize register allocation.

Acknowledgements. This work was partly supported by UK EPSRC project EP/F030592/1 ‘Adaptive Hardware Systems with Novel Algorithmic Design and Guaranteed Resource Bounds’.

References

entity inc_box is
  port ( n_empty : in std_logic;
         n : in std_logic_vector (63 downto 0);
         n_rd : out std_logic;
         r_full : in std_logic;
         r : out std_logic_vector (63 downto 0);
         r_wr : out std_logic;
         nn_full : in std_logic;
         nn : out std_logic_vector (63 downto 0);
         nn_wr : out std_logic;
         clock : in std_logic;
         reset : in std_logic );
end inc_box;

architecture FSM of inc_box is
  type t_state is (R1a, R1b, Ready);
  signal state : t_state ;
begin
  process ( clock , reset )
  variable r1_n : std_logic_vector (63 downto 0);
  begin
    if ( reset = '0' ) then
      state <= Ready ;
      n_rd <= ' 0 ' ;
      r_wr <= ' 0 ' ;
      nn_wr <= ' 0 ' ;
    elsif rising_edge ( clock ) then
      case state is
        when Ready =>
          if n_empty = '0' then
            r1_n := n ;
            n_rd <= ' 1 ' ;
            state <= R1a ;
          end if ;
        when R1a =>
          n_rd <= ' 0 ' ;
          if nn_full = '0' and r_full = '0' then
            nn <= r1_n+1 ;
            nn_wr <= ' 1 ' ;
            r <= r1_n ;
            r_wr <= ' 1 ' ;
            state <= R1b ;
          end if ;
        when R1b =>
          nn_wr <= ' 0 ' ;
          r_wr <= ' 0 ' ;
          state <= Ready ;
        end case ;
      end case ;
    end if ;
  end process ;
end FSM;

Listing 1.1. VHDL code generated for the inc box
Functor: A Distributed Computing Library for Objective Caml *

Jean-Christophe Filliâtre and K. Kalyanasundaram

CNRS, LRI, Univ Paris-Sud 11, Orsay F-91405
INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
filliatr@lri.fr, kalyan.krishnamani@inria.fr

Abstract. We present Functor, a distributed computing library for Objective Caml. The main features of this library include (1) a polymorphic API, (2) several implementations to adapt to different deployment scenarios such as sequential, multi-core or network, and (3) a reliable fault-tolerance mechanism. This paper describes the motivation behind this work, as well as the design and implementation of the library. It also demonstrates the potential of the library using realistic experiments.

1 Introduction

This paper introduces Functor, a generic library for distributed computing for a widely used functional programming language, Objective Caml (OCaml for short). This work was initially motivated by the computing needs that exist in our own research team. Our applications include large-scale deductive program verification, which amounts to checking the validity of a large number of logical formulas using a variety of automated theorem provers [7]. Our computing infrastructure consists of a few powerful multi-core machines (typically 8 to 16 cores) and several desktop PCs (typically dual-core). However, for our application needs, no existing library provides a polymorphic API with usual map/fold higher-order operations, built-in fault-tolerance, and the ability to easily switch between multi-core and network infrastructures. Hence we designed and implemented such a library, which is the subject of this paper. The library is available at http://functor.lri.fr/.

The distributed computing library presented in this paper is not a library that helps in parallelizing computations. Rather, it provides facilities for reliable, distributed execution of parallelizable computations. In particular, it provides a set of user-friendly APIs that allows distributed execution of large-scale parallelizable computations, very relevant to our application needs (and also relevant to a variety of real-world applications). Further, the distributed execution could be over multiple cores in the same machine or over a network of machines. The most important features of our library are the following:

* This research was partly supported by the French national project U3CAT (Unification of Critical C Code Analysis Techniques, ANR-08-SEGI-021).
Genericity: it allows various patterns of polymorphic computations;
Simplicity: switching between multiple cores on the same machine and a network of machines is as simple as changing a couple of lines of code;
Task distribution and fault-tolerance: it provides automatic task distribution and a robust fault-tolerance mechanisms, thereby relieving the user from implementing such routines.

The application domain of such a distributed computing library is manyfold. It serves a variety of users and a wide spectrum of needs, from desktop PCs to networks of machines. Typical applications would involve executing a large number of computation expensive tasks in a resource-optimal and time-efficient manner. This is also the case in our research endeavours, that is validating thousands of verification conditions using automated theorem provers, utilizing the computing infrastructure to the maximum. It is worth noting that Functory is not targeted at applications running on server farms, crunching enormous amounts of data, such as Google’s MapReduce [6].

In the following, we introduce our approach to distributed computing in a functional programming setting and distinguish it from related work.

Distributed Computing. A typical distributed computing library, as Functory, provides the following (we borrow some terminology from Google’s MapReduce):

- A notion of tasks which denote atomic computations to be performed in a distributed manner;
- A set of processes (possibly executing on remote machines) called workers that perform the tasks, producing results;
- A single process called a master which is in charge of distributing the tasks among the workers and managing results produced by the workers.

In addition to the above, distributed computing environments also implement mechanisms for fault-tolerance, efficient storage, and distribution of tasks. This is required to handle network failures that may occur, as well as to optimize the usage of machines in the network. Another concern of importance is the transmission of messages over the network. This requires efficient marshalling of data, that is encoding and decoding of data for transmission over different computing environments. It is desirable to maintain architecture independence while transmitting marshalled data, as machines in a distributed computing environment often run on different hardware architectures and make use of different software platforms. For example, machine word size or endianness may be different across machines on the network.

A Functional Programming Approach. Our work was initially inspired by Google’s MapReduce\(^1\). However, our functional programming environment allows us to be

\(^1\)Ironically, Google’s approach itself was inspired by functional programming primitives.
more generic. The main idea behind our approach is that workers may implement any polymorphic function:

\[ \text{worker: } \alpha \rightarrow \beta \]

where \( \alpha \) denotes the type of tasks and \( \beta \) the type of results. Then the master is a function to handle the results together with a list of initial tasks:

\[ \text{master: } (\alpha \rightarrow \beta \rightarrow \alpha \ \text{list}) \rightarrow \alpha \ \text{list} \rightarrow \text{unit} \]

The function passed to the master is applied whenever a result is available. The first argument is the task (of type \( \alpha \)) and the second one its result (of type \( \beta \)). It may in turn generate new tasks, hence the return type \( \alpha \ \text{list} \). The master is executed as long as there are pending tasks.

Our library makes use of OCaml’s marshalling capabilities as much as possible. Whenever master and worker executables are exactly the same, we can marshal polymorphic values and closures. However, it is not always possible to have master and workers running the same executable. In this case, we cannot marshal closures anymore but we can still marshal polymorphic values as long as the same version of OCaml is used to compile master and workers. When different versions of OCaml are used, we can no longer marshal values but we can still transmit strings between master and workers. Our library adapts to all these situations, by providing several APIs.

**Related Work.** In order to compare and better distinguish Functory from others work with related goals and motivations, we can broadly classify the related work in this domain into:

1. **Distributed Functional Languages (DFLs)** — functional languages that provide built-in primitives for distribution. Examples include ML5, Jo&Caml Glasgow Distributed Haskell, Erlang, etc.

2. **Libraries for existing functional languages** — that could be readily used in order to avoid implementing details like task distribution, fault-tolerance, socket programming, etc.

Functory belongs to the second category. For reasons of completeness, though, we first describe some existing DFLs related to functional programming.

Jo&Caml is one of the DFLs which provides communication primitives (like channels) for facilitating transmission of computations. However, it does not provide ready-made language features for fault-tolerance, which is indispensable in a distributed setting. The user has to include code for fault-tolerance, as already demonstrated in some Jo&Caml library [10]. ML5 [11], a variant of ML, is a programming language for distributed computing, specialized for web programming. It provides primitives for transferring control between the client and the server, as well as low-level primitives for marshalling the data. As in the case before, ML5 is a programming language that offers primitives for code mobility, and the code for distribution of computation and fault-tolerance has to be included by the user. ML5 implements type-safe marshalling and Functory does not, though an existing type-safe marshalling library could be used with Functory. Glasgow
Distributed Haskell (GdH) [13] is a pure distributed functional language that is built on top of Glasgow Haskell and provides features for distributed computing. It is an extension of both Glasgow Parallel Haskell, that supports only one process and multiple threads and Concurrent Haskell that supports multiple processes. It also offers features for fault-tolerance - error detection and error recovery primitives in the language.

CamlP3l [1] mixes the features of functional programming with predefined patterns for parallel computation to offer a parallel programming environment. Again, it is a programming language offering primitives for distributing computation to parallel processes and also to merge the results from parallel executions. Erlang [3] is a programming language which has features for distribution and fault-tolerance. In particular, it has features for task distribution and is more well-known for its rich error detection primitives and the ability to support hot-swapping. The error detection primitives of Erlang allow nodes to monitor processes in other nodes and also facilitate automatic migration of tasks in failed nodes to recovered or active nodes.

Any DFL above could have been used to implement our library. Our motivation, though, was neither to implement our system using any existing DFL nor to come up with a new DFL. The goal of Functory is rather to provide the users of an existing general-purpose functional programming language, namely OCaml, high-level user-friendly APIs that hide the messy details of task distribution and fault-tolerance. We now turn to distributed computing libraries for general purpose functional languages and weed out the distinguishing features of Functory.

There are several implementations of Google’s MapReduce in functional programming languages. But Functory was just inspired by Google’s MapReduce and is not exactly a MapReduce implementation. The simplest difference comes from the very fact that Functory does not operate on key/value pairs. PlasmaMR [2] is an OCaml implementation of Google’s MapReduce on a distributed file system PlasmaFS. It is able to use PlasmaFS to its advantage — the ability of the file system to handle large files and query functions that implement data locality to optimize network traffic. However, PlasmaMR does not support fault-tolerance which is indispensable in any distributed computing application. Another MapReduce implementation in OCaml is Yohann Padioleau’s [12]. It is built on top of OCamlMPI [9], while our approach uses a homemade protocol for message passing. Currently, we have less flexibility w.r.t. deployment of the user program than OCamlMPI; on the other hand, we provide a more generic API together with fault-tolerance.

The iTask system [8] is a library for the functional language ‘Clean’ targeted at distributed workflow management. The library provides a set of combinators (some of which perform map/fold operations) that facilitate applications running in different nodes of a distributed system to communicate, exchange information and coordinate their computations.
2 API

This section describes our API. We start from a simple API which is reduced to a single higher-order polymorphic function. Then we explain how this function is actually implemented in terms of low-level primitives, which are also provided in our API. Conversely, we also explain how the same function can be used to implement high-level distribution functions for map and fold operations. Finally, we explain how our API is implemented in five different ways, according to five different deployment scenarios.

2.1 A Generic Distribution Function

The generic distribution function in our API follows the idea sketched in the introduction. It has the following signature:

```
val compute:
  worker: (α → β) →
  master: (α × γ → β → (α × γ) list) → (α × γ) list → unit
```

Tasks are pairs, of type α × γ, where the first component is passed to the worker and the second component is local to the master. The worker function should be pure\(^2\) and is executed in parallel in all worker processes. The function master, on the contrary, can be impure and is only executed sequentially in the master process. The master function typically stores results in some internal data structure. Additionally, it may produce new tasks, as a list of type (α × γ) list, which are then appended to the current set of pending tasks.

2.2 Low-level Primitives

The function compute above can actually be implemented in terms of low-level primitives, such as adding a task, adding a worker, performing some communication between master and workers, etc. These primitives are provided in our API, such that the user can interact with the execution of the distributed computation. For instance, a monitoring-like application can use these primitives to allow observation and modification of resources (tasks, workers) during the course of a computation. A type for distributed computations is introduced:

```
  type (α, γ) computation
```

A computation is created with a function create, which accepts the same worker and master as compute:

```
val create: worker: (α → β) →
  master: (α × γ → β → (α × γ) list) → (α, γ) computation
```

Contrary to compute, it takes no list of tasks and returns immediately. Tasks can be added later using the following function:

\(^2\) We mean observationally pure here but we allow exceptions to be raised to signal failures.
A function is provided to perform one step of a given computation:

val one_step: (α, γ) computation → unit

Calling this function results in one exchange of messages between master and workers: task assignments to workers, results returned to the master, etc. A few other functions are provided, such as status to query the status of a computation, clear to remove all tasks, etc.

Using these low-level primitives, it is straightforward to implement the compute function. Basically, it is as simple as the following:

let compute ~worker ~master tasks =
  let c = create worker master in
  List.iter (add_task c) tasks;
  while status c = Running do one_step c done

2.3 High-level API

In most cases, the easiest way to parallelize an execution it to make use of operations over lists, where processing of the list elements are done in parallel. To facilitate such a processing, our library provides most commonly used list operations, all implemented using our generic compute function.

The most obvious operation is the traditional map operation over lists, that is

val map: f:(α → β) → α list → β list. Each task consists of the application of function f to a list element. More interesting is a combination of map and fold operations. For instance, we provide different flavors of function

val map_fold: f:(α → β) → fold:(γ → β → γ) → γ → α list → γ

which, given two functions, an accumulator a and a list l, computes

\[
\text{fold...}(\text{fold}(\text{fold}(a \ (f \ x_1))(f \ x_2))...(f \ x_n))
\]

for some permutation \([x_1, x_2, ..., x_n]\) of the list \(l\). We assume that the \(f\) operations are always performed in parallel. Regarding fold operations, we distinguish two cases: either fold operations are computationally less expensive than \(f\) and we perform them locally; or fold operations are computationally expensive and we perform them in parallel. Thus we provide two functions map_local_fold and map_remote_fold.

In the case of map_remote_fold, only one fold operation can be performed at a time (possibly in parallel with \(f\) operations), as obvious from (1). However, there are cases where several fold operations can be performed in parallel, as early as intermediate results of fold operations are available. This is the case when fold is an associative operation (which implies that types \(\beta\) and \(\gamma\) are the same). Whenever fold is also commutative, we can perform even more fold operations in parallel. Thus our API provides two functions map_fold_a and map_fold_ac for these two particular cases, with types
\texttt{val \hspace{1em} map\_fold\_ac, \hspace{1em} map\_fold\_a:}
\texttt{\hspace{1em} f:(\alpha \rightarrow \beta) \rightarrow \text{fold:}(\beta \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha \text{ list} \rightarrow \beta}

It is rather straightforward to derive these five functions from the generic \texttt{compute} function; readers interested in details could refer to the source code.

### 2.4 Deployment Scenarios

Actually, our library provides not just one implementation for the API above, but instead five different implementations depending on the deployment scenario. The first two scenarios are the following:

1. **Purely sequential execution**: this is mostly intended to be a reference implementation for performance comparisons, as well as for debugging;

2. **Several cores on the same machine**: this implementation is intended to distribute the computation over a single machine and it makes use of UNIX processes;

The next three scenarios are intended for distributing the computation over a network of machines.

3. **Same executable run on master and worker machines**: this implementation makes use of the ability to marshal OCaml closures and polymorphic values.

4. **Master and workers are different programs, compiled with the same version of OCaml**: we can no longer marshal closures but we can still marshal polymorphic values. API functions are split into two sets, used to implement master and workers respectively.

5. **Master and workers are different programs, not even compiled with the same version of OCaml**: we can no longer use marshalling, so API functions are restricted to work on strings instead of polymorphic values.

Our library is organized into three modules: \texttt{Sequential} for the pure sequential implementation, \texttt{Cores} for multiple cores on the same machine and \texttt{Network} for a network of machines, respectively. The \texttt{Network} module itself is organized into three sub-modules, called \texttt{Same}, \texttt{Poly} and \texttt{Mono}, corresponding to contexts 3, 4 and 5 above.

### 2.5 Several Libraries in One

From the description above, it is clear that our library provides several APIs of different granularities, as well as several implementations for various deployment scenarios. Most combinations are meaningful, resulting in thirteen possible different ways of using our library. For instance, one may use the low-level API on a single multi-core machine, or use the high-level API on a network of machines all running the same executable, etc. From the implementation point of view, there is almost no code duplication. We are using OCaml functors to derive specific implementations from generic ones.
3 Implementation Details

The implementation of the Sequential module is straightforward and does not require any explanation. The Cores module is implemented with Unix processes, using the fork and wait system calls provided by the Unix library of OCaml. We do not describe this implementation but rather focus on the more interesting module Network.

3.1 Marshaling

As mentioned in Section 2, the Network module actually provides three different implementations, according to three different execution scenarios. There are provided in three sub-modules, as described below.

Same. This module is used when master and workers are running the same executable. The master and workers have to be differentiated in some manner. We use an environment variable WORKER for this purpose. When set, it indicates that the executable acts as a worker. At runtime, a worker immediately enters a loop waiting for tasks from the master, without even getting into the user code. As explained in Section 2, the master function has the following signature.

\[
\text{val compute: worker:(} \alpha \to \beta \text{) } \to \text{ master:(} \alpha \times \gamma \to \beta \to (\alpha \times \gamma) \text{ list) } \to (\alpha \times \gamma) \text{ list } \to \text{ unit}
\]

The master uses marshalling to send both a closure of type \(\alpha \to \beta\) and a task of type \(\alpha\) to the worker. The resulting strings are passed as argument \(f\) and \(x\) in message Assign. Similarly, the worker uses marshalling to send back the result of the computation of type \(\beta\), which is the argument \(s\) in message Completed.

Though the ability to run the same executable helps a lot in deploying the program in different machines, it comes at a small price. Since the worker is not getting into the user code, closures which are transmitted from the master cannot refer to global variables in the user code. Indeed, the initialization code for these global variables is never reached on the worker side. For instance, some code for drawing Mandelbrot’s set could be written as follows:

\[
\text{let max_iterations} = 200
\]

\[
\text{let worker si} = \ldots \text{ draw sub-image si using max_iterations } \ldots
\]

That is, the global function \textbf{worker} makes use of the global variable \texttt{max_iterations}. The worker gets the function to compute from the master, namely the closure corresponding to function \texttt{worker} in that case, but on the worker side the initialization of \texttt{max_iterations} is never executed.

One obvious solution is not to use global variables in the worker code. This is not always possible, though. To overcome this, the Same sub-module also provides a Worker.compute function to start the worker loop manually from the user code. This way, it can be started at any point, in particular after the initialization of the required global variables. Master and worker are still running the
same executable, but are distinguished using a user-defined way (command-line argument, environment variable, etc.).

There are situations where it is not possible to run the same executable for master and workers. For instance, architectures or operating systems could be different across the network. For that reason, the Network module provides two other implementations.

**Poly.** When master and workers are compiled with the same version of OCaml, we can no longer marshal closures but we can still marshal polymorphic values. Indeed, an interesting property of marshalling in OCaml is to be fully architecture-independent, as long as a single version of OCaml is used. It is worth pointing out that absence of marshaled closures now enables the use of two different programs for master and workers. This is not mandatory, though, since master and workers could still be distinguished at runtime as in the previous case.

On the worker side, the main loop is started manually using Worker.compute. The computation to be performed on each task is given as an argument to this function. It thus looks as follows:

\[
\text{Worker.compute: } (\alpha \to \beta) \to \text{unit} \to \text{unit}
\]

On the master side, the compute function is simpler than in the previous case, as it has one argument less, and thus has the following signature.

\[
\text{Master.compute: master:} (\alpha \times \gamma \to \beta \to (\alpha \times \gamma) \text{ list}) \to (\alpha \times \gamma) \text{ list} \to \text{unit}
\]

For realistic applications, where master and workers are completely different programs, possibly written by different teams, this is the module of choice in our library, since it can still pass polymorphic values over the network. The issues of marshalling are automatically taken care of by the OCaml runtime.

The derived API presented in Section 2.3 is adapted to deal with the absence of closures. Exactly as the compute function, each API now takes two forms, one for the master and another for the workers. For example, map_fold_ac takes the following forms.

\[
\text{Worker.map_fold_ac: } f: (\alpha \to \beta) \to \text{fold:} (\beta \to \beta \to \beta) \to \text{unit}
\]

\[
\text{Master.map_fold_ac: } \beta \to \alpha \text{ list} \to \beta
\]

It is the responsibility of the user to ensure consistency between master and workers.

**Mono.** When master and workers are compiled using different versions of OCaml, we can no longer use marshalling. As in the previous case, we split compute into two functions, one for master and one for workers. In addition, values transmitted over the network can only be strings. The signature thus takes the following form.

\[
\text{Worker.compute: } (\text{string} \to \text{string}) \to \text{unit}
\]

\[
\text{Master.compute: master:} (\text{string} \times \gamma \to \text{string} \to (\text{string} \times \gamma) \text{ list}) \to (\text{string} \times \gamma) \text{ list} \to \text{unit}
\]
Any other datatype for tasks should be encoded to/from strings. This conversion is left to the user. Note that the second component of each task is still polymorphic (of type $\gamma$ here), since it is local to the master.

### 3.2 Protocol

The **Network** module implements the distributed computing library for a network of machines. It provides a function $\text{declare-workers} : \text{n: int} \rightarrow \text{string} \rightarrow \text{unit}$ to fill a table of worker machines.

The **Network** module is based on a traditional TCP-based client/server architecture, where each worker is a server and the master is the client of each worker. The main execution loop is similar to the one in the **Cores** module, where distant processes on remote machines correspond to sub-processes and idle cores are the idle cores of remote workers. The master is purely sequential. In particular, when running the user master function, it is not capable of performing any task-related computation. This is not an issue, as we assume the master function not to be time-consuming. The worker, on the other hand, forks a new process to execute the task and hence can communicate with the master during its computation. We subsequently describe issues of message transfer and fault-tolerance.

Messages sent from master to workers could be any of the following kinds:

- **Assign**(id: int, f: string, x: string) This message assigns a new task to the worker, the task being identified by the unique integer id. The task to be performed is given by strings f and x, which are interpreted depending on the context.
- **Kill**(id: int) This message tells the worker to kill the task identified by id.
- **Stop** This message informs the worker about completion of the computation, so that it may choose to exit.
- **Ping** This message is used to check if the worker is still alive, expecting a **Pong** message from the worker in return.

Messages sent by workers could be any of the following kinds:

- **Pong** This message is an acknowledgment for a **Ping** message from the master.
- **Completed**(id: int, s: string) This message indicates the completion of a task identified by id, with result s.
- **Aborted**(id: int) This message informs the master that the task identified by id is aborted, either as a response to a **Kill** message or because of a worker malfunction.

Our implementation of the protocol works across different architectures, so that master and workers could be run on completely different platforms w.r.t. endianness, version of OCaml and operating system.

### 3.3 Fault-Tolerance

The main issue in any distributed computing environment is the ability to handle faults, which is also a distinguishing feature of our library. The fault-tolerance
mechanism of Functory is limited to workers; handling master failures is the responsibility of the user, for instance by periodically logging the master’s state. Worker faults are mainly of two kinds: either a worker is stopped, and possibly later restarted; or a worker is temporarily or permanently unreachable on the network. To provide fault-tolerance, our master implementation is keeping track of the status of each worker. This status is controlled by two timeout parameters $T_1$ and $T_2$ and Ping and Pong messages sent by master and workers, respectively. There are four possible statuses for a worker:

- **not connected**: there is no ongoing TCP connection between the master and the worker;
- **alive**: the worker has sent some message within $T_1$ seconds;
- **pinged**: the worker has not sent any message within $T_1$ seconds and the master has sent the worker a Ping message within $T_2$ seconds;
- **unreachable**: the worker has not yet responded to the Ping message (for more than $T_2$ seconds).

Whenever we receive a message from a worker, its status changes to **alive** and its timeout value is reset.

Fault tolerance is achieved by exploiting the status of workers as follows. First, tasks are only assigned to workers with either **alive** or **pinged** status. Second, whenever a worker executing a task $t$ moves to status **not connected** or **unreachable**, the task $t$ is rescheduled, which means it is put back in the set of pending tasks. Whenever a task is completed, any rescheduled copy of this task is either removed from the set of pending tasks or killed if it was already assigned to another worker.

It is worth noticing that our library is also robust w.r.t. exceptions raised by the user-provided worker function. In that case, an Aborted message is sent to the master and the task is rescheduled. It is the responsibility of the user to handle such exceptions if necessary.

### 4 Experiments

In this section, we demonstrate the potential of using Functory on several case studies. The source code for all these case studies is contained in the distribution, in sub-directory tests/.

The purpose of the following experiments is to compare the various deployments, namely sequential, cores and network. For this comparison to be fair, all computations are performed on the same machine, a 8 core Intel Xeon 3.2 GHz running Debian Linux. The sequential implementation uses a single core. The
multi-core implementation uses up to 8 cores of the machine. The network implementation uses 8 workers running locally and a master running on a remote machine over a LAN (which incurs communication cost).

4.1 N-queens

The first example is the classical $N$-queens problem, where we compute the total number of ways to place $N$ queens on a $N \times N$ chessboard in such a way no two queens attack each other. We use a standard backtracking algorithm for this problem, which places the queens one by one starting from the first row. Distributing the computation is thus quite easy: we consider all possible ways to place queens on the first $D$ rows and then perform the subsequent search in parallel. Choosing $D = 1$ will result in exactly $N$ tasks; choosing $D = 2$ will result in $N^2 - 3N + 2$ tasks; greater values for $D$ would result in too many tasks.

Each task only consists of three integers and its result is one integer, which is the total number of solutions for this task. We make use of function map_local_fold from the derived API, where $f$ is performing the search and fold simply adds the intermediate results. In the network configuration, we make use of the Network.Same module, workers and master being the same executable.

The following table shows execution times for various values of $N$ and our three different implementations: Sequential, Cores, and Network. The purpose of this experiment is to measure the speedup w.r.t. the sequential implementation.

The first column shows the value of $N$. The number of tasks is shown in second column. Then the last three columns show execution times in seconds for the three implementations. The figures within brackets show the speedup w.r.t. sequential implementation.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$D$</th>
<th>#tasks</th>
<th>Sequential</th>
<th>Cores (7.45×)</th>
<th>Network (6.47×)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1</td>
<td>16</td>
<td>15.2</td>
<td>2.04</td>
<td>2.35</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>210</td>
<td>15.2</td>
<td>2.01</td>
<td>21.80</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>17</td>
<td>107.0</td>
<td>17.20</td>
<td>16.20</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>240</td>
<td>107.0</td>
<td>104.00</td>
<td>24.90</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>18</td>
<td>787.0</td>
<td>123.00</td>
<td>125.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>272</td>
<td>787.0</td>
<td>103.00</td>
<td>124.00</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>19</td>
<td>6120.0</td>
<td>937.00</td>
<td>940.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>306</td>
<td>6120.0</td>
<td>796.00</td>
<td>819.00</td>
</tr>
</tbody>
</table>

From the table above, it is clear that the Cores and Network implementations provide a significant speedup. As evident from the last row, the speedup is almost 8, which is also the number of cores we use. It is also evident from the last column that the Network implementation performs significantly better when the computation time dominates in the total execution time. The two extreme cases correspond to the second and the last row: in the second row, the communication time dominates and is in fact more than 91% of the total execution time; on the other hand, for the last row communication time amounts to just 4.6% of the total execution time. As expected, the network implementation is only beneficial
when the computation time for each individual task is significant, which is the case in realistic examples.

4.2 Matrix Multiplication

This benchmark was inspired by the PASCO’10 programming contest [5]. It consists of multiplication of two square matrices of dimension 100 with integer coefficients. Coefficients have several thousands of digits, hence we use GMP [4] to handle operations over coefficients.

We compare the performances of two different implementations. In the first one, called \texttt{mm1}, each task consists of the computation of a single coefficient of the resultant matrix. In the second one, called \texttt{mm2}, each task consists of the computation of a whole row of the resultant matrix. As a consequence, the total number of tasks in 10,000 for \texttt{mm1} and only 100 for \texttt{mm2}. On the contrary, each task result for \texttt{mm1} is a single integer, while for \texttt{mm2} it is a row of 100 integers.

The experimental results (in seconds) are tabulated below.

<table>
<thead>
<tr>
<th></th>
<th>\texttt{mm1} (10,000 tasks)</th>
<th>\texttt{mm2} (100 tasks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>20.3</td>
<td>20.2</td>
</tr>
<tr>
<td>Cores</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2 cores)</td>
<td>22.7 (0.89×)</td>
<td>11.3 (1.79×)</td>
</tr>
<tr>
<td>(4 cores)</td>
<td>12.3 (1.65×)</td>
<td>6.1 (3.31×)</td>
</tr>
<tr>
<td>(6 cores)</td>
<td>8.6 (2.36×)</td>
<td>4.3 (4.70×)</td>
</tr>
<tr>
<td>(8 cores)</td>
<td>8.0 (2.54×)</td>
<td>3.5 (5.77×)</td>
</tr>
</tbody>
</table>

The difference in the number of tasks explains the differences in the speedup ratios above. We do not include results for the network configuration, as they do not achieve any benefit with respect to the sequential implementation. The reason is that the communication cost dominates the computation cost in such a way that the total execution time is always greater than 30 seconds. Indeed, irrespective of the implementation (\texttt{mm1} or \texttt{mm2}), the total size of the transmitted data is $10^6$ integers, which in our case amounts to billions of bytes.

A less naive implementation would have the worker read the input matrices only once, e.g. from a file, and then have the master send only row and column indices. This would reduce the amount of transmitted data to 10,000 integers only.

4.3 Mandelbrot Set

Drawing the Mandelbrot set is another classical example that could be distributed easily, since the color of each point can be computed independently of the others. This benchmark consists in drawing the fragment of the Mandelbrot set with lower left corner $(-1.1, 0.2)$ and upper right corner $(-0.8, 0.4)$, as a $9,000 \times 6,000$ image. If the total number of tasks $t \geq 1$ is given as a parameter, it is immediate to split the image into $t$ sub-images, each of which is computed in parallel with and independently of the others. In our case, the image is split
into horizontal slices. Each task is thus four floating-point numbers denoting the region coordinates, together with two integers denoting the dimensions of the sub-image to be drawn. The result of the task is a matrix of pixels, of size 54,000,000/t. For instance, using \( t = 20 \) tasks will result in 20 sub-images of size 10.3 Mb each, assuming each pixel is encoded in four bytes.

The sequential computation of this image consumes 29.4 seconds. For \textit{Cores} and \textit{Network} implementations, the computation times in seconds are tabulated below.

<table>
<thead>
<tr>
<th>#cores</th>
<th>#tasks</th>
<th>Cores</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>15.8 (1.86\times)</td>
<td>20.3 (1.45\times)</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>15.7 (1.87\times)</td>
<td>18.7 (1.57\times)</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>16.1 (1.83\times)</td>
<td>19.8 (1.48\times)</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>19.6 (1.50\times)</td>
<td>38.6 (0.76\times)</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>9.50 (3.09\times)</td>
<td>14.4 (2.04\times)</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>8.26 (3.56\times)</td>
<td>11.4 (2.58\times)</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>8.37 (3.51\times)</td>
<td>11.4 (2.58\times)</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>10.6 (2.77\times)</td>
<td>20.5 (1.43\times)</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>9.40 (3.13\times)</td>
<td>12.6 (2.33\times)</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>4.24 (6.93\times)</td>
<td>7.6 (3.87\times)</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>4.38 (6.71\times)</td>
<td>7.5 (3.92\times)</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>6.86 (4.29\times)</td>
<td>11.3 (2.60\times)</td>
</tr>
</tbody>
</table>

The best timings are achieved for the \textit{Cores} configuration, where communications happen within the same machine and are thus cheaper. There are two significant differences with respect to the n-queens benchmark. On one hand, the number of tasks can be controlled more easily than in the case of n-queens. We experimentally figured out the optimal number of tasks to be 30. One the other hand, each computation result is an image, rather than just an integer as in the case of n-queens. Consequently, communication costs are much greater. In this particular experiment, the total size of the results transmitted is more than 200 Mb.

### 4.4 SMT Solvers

Here we demonstrate the potential of our library for our application needs as mentioned in the introduction. We consider 80 challenging verification conditions (VC) obtained from the Why platform [7]. Each VC is stored in a file, which is accessible over NFS. The purpose of the experiment is to check the validity of each VC using several automated provers (namely Alt-Ergo, Simplify, Z3 and CVC3).

The master program proceeds by reading the file names, turning them into tasks by multiplying them by the number of provers, resulting in 320 tasks in total. Each worker in turn invokes the given prover on the given file, within a timeout limit of 1 minute. Each task completes with one of the four possible outcomes: \textit{valid}, \textit{unknown} (depending on whether the VC is valid or undecided...
by the prover), timeout and failure. The result of each computation is a pair denoting the status and the time spent in the prover call. The master collects these results and sums up the timings for each prover and each possible status.

Our computing infrastructure for this experiment consists of 3 machines with 4, 8 and 8 cores respectively, the master being run on a fourth machine. The figure below shows the total time in minutes spent by each prover for each possible outcome.

<table>
<thead>
<tr>
<th>prover</th>
<th>valid</th>
<th>unknown</th>
<th>timeout</th>
<th>failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt-ergo</td>
<td>406.0</td>
<td>3.0</td>
<td>11400.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Simplify</td>
<td>0.5</td>
<td>0.4</td>
<td>1200.0</td>
<td>222.0</td>
</tr>
<tr>
<td>Z3</td>
<td>80.7</td>
<td>0.0</td>
<td>1800.0</td>
<td>1695.0</td>
</tr>
<tr>
<td>CVC3</td>
<td>303.0</td>
<td>82.7</td>
<td>4200.0</td>
<td>659.0</td>
</tr>
</tbody>
</table>

These figures sum up to more than 6 hours if provers were executed sequentially. However, using our library and our 3-machine infrastructure, it completes in 22 minutes and 37 seconds, giving us a speedup of more than 16×. We are still far away from the ideal ratio of 20× (we are using 20 cores), since some provers are allocating a lot of memory and time spent in system calls is not accounted for in the total observed time. However, a ratio of 16× is already a significant improvement for our day-to-day experiments.

5 Conclusions and Future Work

In this paper, we presented a distributed programming library for OCaml. The main features are the genericity of the interface, which makes use of polymorphic higher-order functions, and the ability to easily switch between sequential, multi-core, and network implementations. In particular, Functory allows to use the same executable for master and workers, which makes the deployment of small programs immediate — master and workers being only distinguished by an environment variable. Functory also allows master and workers to be completely different programs, which is ideal for large scale deployment. Another distinguishing feature of our library is a robust fault-tolerance mechanism which relieves the user of cumbersome implementation details. Yet another interesting feature of the library is the ability to add workers dynamically. Functory also allows to cascade several distributed computations inside the same program. Finally, the low-level API of Functory can be used to write interactive programs.

Future Work. There are still some interesting features that could be added to our library.

- One is the ability to efficiently assign tasks to workers depending on resource parameters, such as data locality, CPU power, memory, etc. This could be achieved by providing the user with the means to control task scheduling. This would enable Functory to scale up to MapReduce-like applications.
Currently, without any information about the tasks, the scheduling is completely arbitrary. In both Cores and Network modules, we use traditional queues for the pending tasks; in particular, new tasks produced by the master are appended to the end of the queue.

- Our library provides limited support for retrieving real-time information about computations and communications. Processing and storing information about workers and tasks locally in the master is straightforward.
- One very nice feature of Google’s MapReduce is the possibility to use redundantly several idle workers on the same tasks for speedup when reaching the end of computation. Since we already have the fault-tolerance implemented, this optimization should be straightforward to add to our library.

We intend to enrich our library with all above features.

Acknowledgments. The authors are grateful to the ProVal team for support and comments on early versions of the library and of this paper.

References

An Application of D-Clean
Distributed Functional Programming
for the Navigation of Robots *

Barnabás Králik, Viktória Zsók, and Zoltán Istenes

Eötvös Loránd University, Faculty of Informatics
H-1117 Budapest, Pázmány Péter sétány 1/C
{kralikba,zsv,istenes}@inf.elte.hu

Abstract. This paper provides D-Clean distributed functional implementation of a multiple target path finding algorithms over a bitmap encoding obstacles to be avoided by robots. The goal is to calculate paths between the starting points of mobile robots and the multiple targets. We represent the bitmaps as optimal quadtrees. A quadtree is a tree data structure that partitions a finite two dimensional space. Using quadtrees multiple paths are extracted with considerable speed-up. The paths finding algorithms are implemented as farm skeletons. Bitmap partitions are processed in distributed way into optimal quadtrees. Adjacency graphs are built for path planning between distinguished points assisting the navigation of robots.

Keywords: D-Clean, distributed graph algorithms, coordination language, robot navigation, path finding, quadtree

1 Introduction

Distributed Clean or D-Clean [10, 18] is a distributed extension for Clean [13]. Our multi-layered distributed system was designed to coordinate functional programming tasks on a cluster using our high level coordination language parameterized by computation nodes of functional programs.

The computation nodes (boxes) are distributed automatically over the grid by the middleware system. The programmer describes the distribution at the highest level using D-Clean expressions, which are transformed into D-Box computation nodes and communication channels. D-Clean is based on the functional composition of coordination primitives, which operate on dataflows. This enables to build skeletons, and compound coordination schemes in a straightforward way.

Widely used skeletons (like farm, divide and conquer, pipe and reduce) have been implemented and measurements demonstrated high speed-ups for more complex problems.

Distributed graph algorithms due to their applicability in artificial intelligence are widely implemented in several distributed systems. Here we describe

* Supported by TÁMOP-4.2.1/B-09/1/KMR-2010-0003
two distributed algorithms for finding many paths on large two-tone bitmaps using the A* heuristic path finding algorithm. The bitmaps are represented by quadtrees (see section 3).

Our first algorithm searches paths using the adjacency graph of the quadtree representation of the bitmap. First, partitions of the bitmap are distributed to different computational threads to build up their corresponding quadtrees. Therefore several quadtrees and their adjacency graphs are created in parallel. Afterwards, they are joined into a final adjacency graph of the original, complete bitmap (see figure 2). The resulted graph is used for the computation of paths in parallel using an A* heuristic path finding algorithm.

The second algorithm finds concurrently paths on different partitions of the bitmap. As in the first algorithm, quadtrees are built in a distributed way for each of the partitions. We modify the path finding method from the first algorithm by introducing a preprocessing step. After creating the quadrees of the partitions and their adjacency graphs, the shortest paths between each pair of boundary points are computed. These shortest paths are joined into a graph describing the structure of the complete map. The A* algorithm is used in order to obtain the final path of the robot on the complete bitmap (details in section 3).

The paper is structured as follows: section 2 presents a short overview of the D-Clean language, section 3 describes how to build the quadtree representation of the bitmap and the path extraction in the two algorithms mentioned above. Section 4 enumerates related works, and finally section 5 concludes.

2 Overview of D-Clean

D-Clean is a language extension of the lazy functional language Clean with distributed language primitives.

D-Clean program coordinates dynamic work distribution over a cluster controlling the distributed computation in the generated computation boxes and communication channels.

A coordination primitive usually has two parameters: a function expression (or a list of function expressions) and a sequence of input channels. The coordination primitives return the result sequence on the output channels. The signature of the coordination primitive, i.e. the types of the input and output channels are inferred according to the type of the embedded Clean expressions.

Each language element uses channels for receiving the input data required for the arguments of their function expression parameters. The results of the function expression are sent to the output channels. Every channel is carrying data elements of a specified base type from one computational node to another one.

Every D-Clean program contains a DStart primitive to start the distributed computation by producing the input data for the dataflow graph. It has no input channels, only output channels. DStop is another coordination primitive which must be included in any D-Clean program. The task of this primitive is to receive and save the result of the computation.
**DMap** is the distributed version of the standard **map** library function, it needs an elementwise processable function as parameter.

The **DApply** coordination primitive is the most general one. It has several variants (see description in [10]). The first variant of **DApply** applies the same function expression \(n\) times on \(n \times k\) channels. The second variant of **DApply** may apply different function expressions, which are given in a function sequence. The types and the number of the arguments of the function expressions can also be different. If the function sequence contains an identity function, then the data received via the corresponding channel is forwarded directly to the output channel and afterwards to the next node.

The **DReduce** and the **DProduce** primitives are special cases of **DApply** with some restrictions. A valid expression for **DReduce** has to reduce the dimension of the input channel type, while the expression of **DProduce** has to increase the dimension of the channel type.

**DDivideS** is a static divider and splits the input data into \(n\) parts and broadcasts them to \(n\) computational nodes. This primitive is called a static divider since the value of \(n\) must be known at pre-compile time.

The counterpart of the **DDivideS** is the **DMerge** primitive. **DMerge** collects the input sublists from channels and builds up the output data lists. All the input channels must have the same type.

The **D-Clean** language is used to distribute well defined sub tasks among several computation nodes in order to achieve a high speed-up for when running the program on large dataflows.

In the following we use the distributed functional D-Clean system for calculating the shortest paths on large two-tone bitmaps for navigation of robots.

## 3 An algorithm for calculating shortest paths on large two-tone bitmaps for robots

### 3.1 Distributed algorithms for robots

The navigation of robots is one of the most challenging problem in the field of mobile robotics. A map with well defined representation is used to calculate the path for the robot. However the maps can be encoded in several different ways: metric, topological, bitmap, quadtree, free space map, object map, composite map. A map is used for navigation purposes to find the path of the robots (shortest, fastest, safest, etc.). Classical graph search algorithms can be applied to find paths (e.g. A* heuristic search algorithm).

As in many fields of informatics, paradigm changes have occurred also in robotics: simple, sequential computation are replaced by parallel distributed ones. Parallelism is encountered at different level in robotics: robot swarms, parallel robot control algorithms, and distributed signal processing.

Robotics uses many algorithms rooted in artificial intelligence, which are very sensitive to the input data size and they risk combinatorial explosions. Real time mobile robotics and changing robot environment also require short response time.
Whenever sequential algorithms are not providing solutions for the above mentioned problems, distributed, parallel approach might solve them. This paper describes two solution for robot navigation by distributed path planning implemented in D-Clean.

Robots in our approach are using a bitmap for orientation and movement between obstacles. Navigation of robots is especially interesting when the bitmap and path planning are processed in distributed way. As the path computation is processor-intensive even for relatively small data sets, we distribute it to multiple computation nodes to acquire low response times to a dynamically changing environment.

We designed and implemented two algorithms using the D-Clean distributed environment for partitioning the bitmap and finding paths in parallel way.

3.2 Parallel quadtree construction

Suppose we have already acquired information about the environment surrounding the mobile robots. The map is represented as a large two-dimensional bitmap of discrete values: passable and obstructed. A broad set of coordinates have also been marked as targets-to-be-examined. The targets are points to be reached by the robots. Our goal is to calculate approximate shortest paths between the starting points of the mobile robots and multiple targets. We call all these points distinguished points.

We focus on a quadtree approach to the problem [7]. A quadtree is a tree data structure for partitioning a set of points in a two-dimensional space.

In a quadtree each node of the tree covers a rectangular partition of the bitmap, the root node itself covering the whole bitmap. Internal nodes partition the matching slice into four quadrants - NE, SE, SW, NW in a map analogy - so that each of these partitions are covered by a corresponding child node. Leaf nodes cover a partition of the bitmap that contains either passable or obstructed values. The leaf is labeled "passable" or "obstructed" accordingly. Let any node in a quadtree and its corresponding rectangle thus be called a quad.
The bitmap image used for the navigation of robots is converted into an optimal quadtree representation. We call a quadtree optimal if each of the internal nodes cover a mixed partition of the bitmap - that is, the corresponding slice contains passable and obstructed values too. This quadtree can be called optimal because its number of vertices is the lowest of all quadtrees that describe the original bitmap.

Defining a neighbor-of relation between the nodes of a quadtree is straightforward. We call two quads neighboring if their rectangles are neighboring, i.e. their union is one consistent polygon.

The quadtree is an index structure of the bitmap that allows for the quick computation of a node’s neighbors.

The adjacency graph of a quadtree is thus a graph in which vertices are labeled with quads and vertices of adjacent quads are connected. Our path finding algorithms process such adjacency graphs that contain the passable leaf nodes of a quadtree.

![Quadtree partitioning of an arbitrary scaled bitmap and the corresponding generated adjacency graph](image)

**Fig. 2.** Quadtree partitioning of an arbitrary scaled bitmap and the corresponding generated adjacency graph

Our first algorithm consists of the following major phases:

1. The master divides the space into equally-sized rectangles and distributes these slices to the workers.
2. Each worker builds the optimal quadtree of its own slice.
3. An adjacency graph is built from each of these quadtrees, so that:
   a. the graph has a vertex for each passable leaf node;
   b. the graph has a vertex for each distinguished point;
   c. for each neighboring leaf node pair the appropriate vertex pair is connected by an undirected edge. The weight of such an edge is the Euclidean distance between the covered rectangles’ center points;
   d. the vertex corresponding to a leaf node, the rectangle of which contains a distinguished point, is connected to the vertex of the distinguished point.
4. A process joins these subgraphs into a $G$ graph. The graph $G$ is equivalent to the adjacency graph of a non-optimal quadtree of the complete bitmap. Each pair of vertices that are labeled with tangent rectangles are connected by an edge. The weight of such an edge is the distance between the centers of these rectangles. The complete adjacency graph is distributed to the workers.

5. Pairs of distinguished points are distributed to workers. Each worker calculates the shortest path between these pairs of distinguished points using a version of the A* heuristic search algorithm.

6. The master collects all the calculated paths.

The algorithm has been implemented using the D-Clean distributed environment. The skeleton is visualized in Figure 3.

![Fig. 3. The computational graph of the first algorithm](image)

The subsequent D-Clean code snippet describes the coordination of the workers:

```plaintext
DStart DDivideS DApply DApply DMerge DApply DDivideS DApply DMerge DStop
slices splits quadTree adjGraph joinGraphs pointPairs splitp findPaths collect save

DistrStart = DStop save DMerge collect
    DApply findPath DDivideS splitp n
    DApply pointPairs DMerge joinGraphs
    DApply adjGraph DApply quadTree
    DDivideS splits n DStart slices
```

The `slices` are cut from the original bitmap, `splits` transfers them to the workers. The `quadTree` function partitions a slice into an optimal quadtree, while `adjGraph` calculates the weighted adjacency graph. The `joinGraphs` function merges the subgraphs while connecting them to each other at neighboring peripheral points. The `pointPairs` function generates distinguished point pairs, `splitp` distributes them. Finally, `findPath` finds a path using a functional version of the A* heuristic search algorithm, `collect` merges these paths into a final data set written by `save` into a format recognizable by the mobile robots.
3.3 Precalculated shortest paths

The second algorithm has an additional distributed pre-processing step. As in the first algorithm, the bitmap is divided into equally sized rectangles and distributed to the workers which build quadtrees of the bitmap.

At this point the construction of the algorithm diverges from the first one’s. A new DApply box is inserted into the pipeline, which calculates shortest paths in the worker’s slices as a preprocessing step.

Further on a master process joins these subgraphs. The greater graph is distributed and used for the final pathfinding.

The major steps of this algorithm are thus as follows:

1. The master divides the space into equally-sized rectangles and distributes these slices to the workers.
2. Each worker builds the optimal quadtree of its own slice.
3. An adjacency graph is built from each of these quadtrees, so that every condition of 3. of the first algorithm hold.
4. Vertices in the adjacency graph that lie on the boundary of the slice are marked "boundary". All shortest paths from boundary points and distinguished points are calculated using an implementation of Dijkstra’s algorithm. The Euclidean distance of the endpoint’s positions is considered the cost of an edge. The position of a quad is said to be its centerpoint.
5. A nearly complete graph is built, so that:
   a. the graph has a vertex for each boundary node;
   b. the graph has a vertex for each distinguished point;
   c. each vertex pair is connected by an undirected edge that is labeled with the list of vertices of the shortest path between the corresponding vertices in the adjacency graph if such a path exists.
6. A process joins these subgraphs into a higher-level graph describing paths in the adjacency graph of the whole bitmap. The edges in this graph are labeled with paths in the adjacency graph. The total cost such a path are considered the weight of the labeled edge. Each pair of vertices \( u, v \) that are labeled with tangent rectangles are connected by an edge labeled with the direct path \( uv \). This graph is distributed to the workers.
7. Pairs of distinguished points are distributed to workers. Each worker calculates the shortest path between these pairs of distinguished points using a version of the A* algorithm.
8. The master collects all calculated paths. Each of these paths are labeled with paths in the adjacency graph of the complete bitmap. The concatenation of these labels gives the path to be taken by the robot.

This algorithm has also been implemented in the D-Clean environment in a way similar to the prior solution. The computational graph is visualized in Figure 4.

The subsequent D-Clean code snippet describes the coordination.
The slices, splits, quadTree and adjGraph functions are exactly same as in the first algorithm; adjGraph has the responsibility of marking boundary nodes as mentioned in step 4. The dijkstra function calculates shortest paths in the workers’ subgraphs. These are merged by joinGraphs. Function pointPairs generates distinguished point pairs; splitp distributes them to workers. Afterwards, findPath finds a path using the A* algorithm, collect merges these into a final data set. The paths in the higher-level graph are then expanded into a sequence of concrete coordinates and written to disk by save.

4 Related work

Distributed functional programming with a different kind of parallelization are present in several different dialects of Haskell (e.g. Eden [1], [11]) or ML (e.g. JoCaML [8]). They are based on different inherent process definitions, while our D-Clean approach uses individually executable well separated computation nodes (boxes) distributed over a grid.

The literature of functional programming skeletons includes two main textbooks. [9] contains functional programming skeleton applications, while [14] is a theoretical approach of the topic. D-Clean is used for distributed skeleton application on clusters.

Earlier parallel quadtree construction and manipulation were implemented and tested on hypercube multiprocessors [16] and transputers. Our implementation focuses on processing and transforming the source bitmap using quadtrees by distributed computation for grid systems.

Various AI algorithms are available that solve the problem in reasonable time (e.g. [16]); however, most of them are inappropriate for implementing as distributed purely functional programs.
Mobile robot path planning among weighted regions using quadtree representation is present in [17] where not two-tone (passable or not passable bitmap) maps are used, but weight parameters are associated with the regions of maps corresponding their difficulty to travel through. These weights are represented in modified quadtrees, and the technique of distance transform is extended to the weighted regions and applied to the robot path planning.

Robot navigation in dynamic environment using quadtrees and parallel processing is described in details in [3].

The standard path calculated using a quadtree is not guaranteed to be the shortest Euclidean path, because of the nature of the quadtree. A new data structure called framed quadtree is proposed to overcome this problem [4]. This approach leads to shorter paths. In robotics quadtrees were used for example in [12].

Functional graph algorithms can be found in the following textbooks: [2] and [15]. The Functional Graph Library ([6]) is a major implementation of functional graph algorithms ([5]) in Haskell and ML. We have implemented in Clean graph representation algorithms based on the above literature.

5 Conclusion and future work

In this paper we presented the quadtree representation of a bitmap used for robot navigation. The quadtree provided an efficient partitioning of the two-tone map from which multiple paths were extracted. The optimal quadtree representation enabled building up a considerably smaller graph for path finding between distinguished points. The algorithm was implemented as a specialization of the farm skeleton for the distributed processing of the bitmap containing robot obstacles. We exploited the inherent parallelism and problem regularity of the bitmap partitioning, therefore we obtained an overall computation efficiency of the algorithms, since we applied the algorithms on relatively structured domain.

We plan to extend the D-Clean system with more flexible computation graph building in order to allow the implementation of new classes of distributed algorithms (e.g. the dynamic wavefront path finding algorithm).

References

Abstract. The world of finance today faces daunting complexity and increasing demand for performance of risk analysis and other simulation software. Simultaneously, new EU-wide transparency and stability rules require considerably more code and system inspection in the financial industry. In this paper, we report about a new project which approaches these problems in a combined novel approach, integrating financial mathematics, domain specific language technology, parallel functional programming, and massively parallel novel hardware.

Our research has potential to generally bring forward the standard in functional programming and render research in parallel programming more goal-directed. Research in HIPERFIT will accumulate expertise in different scientific fields and foster synergies between application area and programming language research, delivering a good showcase of modern language technology and advanced functional methods in an area of increasing importance.

1 Introduction

Today, the financial sector is confronted with fundamental computational challenges: Data volumes to be handled are growing at an exponential rate; stochastic simulations consume in principle limitless numbers of compute cycles; quantitative and auditable risk management is becoming mandatory; real-time requirements hit speed-of-light limitations. At the same time, custom OTC contracts require more expressive modelling languages and development cycles become shorter than even imagined five years ago. This requires complex computational
models, specifications and systems that are guaranteed to be meaningful, transparent, rapidly developed, and scalable on today’s and tomorrow’s hardware. What makes this a fundamentally new and interesting scientific challenge is that the problems need to be solved simultaneously and thus trade-offs between the underlying financial mathematics, problem modeling, programming language technology, high-performance systems and practical applicability must be explicitly accounted for.

To address these problems, we are currently establishing the Center for Functional High-Performance Computing for Financial Information Technology (HIPERFIT), an internationally unique strategic research center which brings together key researchers in exactly the required scientific fields – programming languages, parallel systems, and mathematical finance – with the Danish banking sector. Our hypothesis is that it is possible and easier to solve the simultaneous challenges of high transparency, high computational performance and high productivity by declarative domain-specific and high-level functional programming languages. The approach taken by HIPERFIT promises to eliminate imperative low-level programming, expressing domain-specific problems directly and mapping them directly to the massively parallel novel hardware. The center closely cooperates with practitioners from Danish banks and financial companies, which provides a touchstone for results and keeps our research focused. Hosted by the Department of Computer Science, the research center involves three departments of the University of Copenhagen and six funding partners from the Danish financial industry. The center is also open for new collaborators.

In the project paper at hand, we present the research paradigm, strategy and organisation of HIPERFIT. We underline the relevance of HIPERFIT’s topics in the following section, and then explain the integrated approach we take with the research center, and research themes we will work on (Section 3). Section 4 provides some technical content for an audience of functional programmers. We summarise the state of art in language support for financial applications (Section 4.1) and give an overview of parallel functional programming paradigms and trends (Sections 4.2 and 4.3). In the following Section 5, we outline two first project activities within HIPERFIT in relation to functional programming. Section 6 concludes.

2 Motivation and Background

Worldwide financial markets become more and more complex, and their effects consequently less and less predictable. In the year 2008, we have seen one of the most severe worldwide financial crises ever. Induced by defaults in the American real-estate market (sub-prime loans), some investment banks collapsed and a large number of globally operating others were affected – taking down a large number of other industries and ultimately leading to a general economic crisis of global scale [15]. The crisis in 2008 demonstrates how complex dependencies are built up in the financial industry and that experts can vastly misjudge the impact of a local crash on other institutes and sectors.
2.1 Need for More Accurate Modelling in Mathematical Finance

To help avoid a repeat of the 2008 crash, financial institutions have initiated internal activities at a massive scale. Huge sums are invested in computational methods for the banks in order to improve modeling financial phenomena with all concerned parties. While the banks already have extensive modeling and pricing activities, the new problems establish a modeling and simulation paradigm vastly different from the existing system. Existing systems are based on macroscopic models and parameterized representations of the individual commitments. The new requirement will be a detailed system of microeconomic models of the individual businesses and the combination of these into a global economic barometer that identifies the value and risk in a given bank.

2.2 Increasing Regulation Requirements

The financial crisis that hit the world economy in 2008 has also triggered several new legislations that seek to govern the financial sector more carefully. As an example of the challenges facing the financial sector, the consequences of current and forthcoming capital adequacy and transparency requirements (Basel-II as implemented in the EU via the Capital Requirements Directive; Basel-III, which is presently under preparation as CRD II-IV; recently proposed SEC rules for computational models of securities) [36] carry enormous requirements on IT systems at all levels, ranging from high-level modeling of financial instruments to auditable internal risk models and their reliable implementation.

2.3 Focus on Numerics and Performance

Quantitative analyses in the financial industry have always called on great computing power. However, such analyses were usually done by so-called “quants”, having a background in mathematical finance, financial engineering, mathematics and physics. Their expertise was in the fields of option pricing, calibration, simulation, stochastic differential equations, partial differential equations, and statistics. Only recently do we see increased focus on the efficiency and transparency of numerical and computational technology used in the quantitative analyses (and increasingly Monte-Carlo and other simulation techniques [17]). Reasons for this trend lie both inside the industry, through an ever-growing competition for achieving more and more marginal benefits, and outside, by imposing new auditing and solvency procedures from international regulation. In consequence, the demand for computing specialists within the domain of mathematical finance has increased. More and more academics and practitioners work with computational and numerical finance aspects with focus on computing methodology and quality rather than more principal financial questions.

3 The Hiperfit Center

Funded by the Danish Council for Strategic Research, the Hiperfit center started its work in January 2011. The center includes four main research ar-
eas involving three different departments of the University of Copenhagen, and initially six funding partners from the Danish financial industry. The publicly funded academic part of HIPERFIT amounts to 31.4M DKK (ca. 4.2M €), creating 6 Ph.D- and 3 post-doctoral positions in different disciplines. Industry partners and departments contribute resources of another 12M DKK (ca. 1.6M €). Within its lifespan of six years, HIPERFIT will contribute strategic research in tailor-made expressive programming languages, frameworks, tools and technologies for financial modeling, and effective use of modern parallel hardware without compromising correctness, transparency or portability.

### 3.1 Integrated Approach to Foster Synergies

Research in HIPERFIT aims to solve problems of today’s computing in finance in a holistic integrated approach. HIPERFIT therefore joins researchers with state-of-the-science expertise in all research areas relevant for high-performance financial applications: Theory and practice of mathematical finance (MF), domain-specific languages (DSLs), functional programming (FP), and high-performance systems (HPS). Their work is organized in general research themes and concrete practical projects. Figure 1 depicts these areas of expertise, and their relationship to our initial research themes.

A rather different trend is what we observe today in a number of scientific and application areas, relating to parallel scientific computing and novel hardware: Excited by the tremendous parallel computing power of modern General-Purpose Graphics Processing Units (GPGPUs, or just GPUs), domain experts dive into classical imperative programming and encode their algorithms in low-level languages with a high degree of platform dependency. While such low-level code written directly by the domain expert might find success in short term, HIPERFIT explicitly aims to avoid such development and over-specialisation, in favour of a more enduring development process and maintainable systems.

---

**Fig. 1.** Relationship between research areas and research themes
One essential ingredient in achieving this is our commitment to declarative languages and advanced compilation of domain specific abstractions. Side-stepping imperative programming bears the elements of a disruptive technology with drastic productivity and performance improvement potential. It requires the traditionally disconnected research communities of mathematical finance, programming languages and high-performance systems, and application experts from the financial sector, to bring cutting edge knowledge from within their respective fields to bear on problems transcending their own field of expertise.

### 3.2 Research Organisation and Methodology

HIPERFIT follows a paradigm of project-driven research theme exploration. Projects are concrete cases for exploration and development, either motivated by practical needs of industry partners (problem-driven), or by the intent to evaluate novel technologies and gather know-how for later use (technology driven). Projects may or may not contain information protected by industry partners. Experience and lessons from our work on projects directs our work on research themes, cross-cutting several research areas within computing for finance, generally leading to open publishable results. Projects help identify useful and timely goals, while research themes provide the headroom and freedom which is necessary for true innovation.

### 3.3 Research Themes in HIPERFIT

Exploratory discussions with our industry partners have led to identifying several cross-cutting research themes for the start of HIPERFIT, depicted in Figure 1. Each research theme will be backed up with project activities, part of which are input by the industry partners.

- **Risk Scenarios** We try to describe the transition from observables (like current prices and historical data) to scenario generation and from scenario generation to reporting and management. Adequate risk scenarios have immediate relevance for management decisions, including deriving capital requirements to ensure stability in unlikely and extreme situations.

- **Model Specification** Financial models in practical use today vary from so-called “model-free” evaluation (prices given completely in terms of other prices) to sophisticated stochastic processes (such as advanced multi-dimensional jump-diffusions). We want to systematically explore and compare benefits and costs of models for different applications (solvency, accounting, or management), including their optimisation potential and implied imprecision on parallel hardware.
**DSLs for Financial Information** Declarative domain-specific languages (DSLs) to describe a range of financial products are already in widespread use in the financial sector today. We aim to complement these languages with similarly expressive DSLs for other financial information, and especially for financial models. Our goal is a complete DSL framework with broad application coverage, suitable both for internal reporting and statistics, external auditing, and computation in large risk scenarios. We will describe the DSL approach in the financial domain, and our goals, in more detail in Section 4.1.

**Extracting Parallelism from Declarative Specifications** The core theme of this research theme is to analyse and transform large-scale financial computations to expose their inherent parallelism. Departing from work on existing applications and domain specific abstractions, we plan to derive a tailor-made language to express large-scale numeric computations in a way natural for mathematical finance, while efficiently executable on modern parallel hardware. Thanks to their high-level nature, parallel functional languages appear to be an excellent platform for this. We expect vector and matrix operations and accumulating reductions to be the major source of parallelism at this stage, but aim to identify more domain-specific parallelisation schemes. The DSL development for financial models will lead to additional or modified requirements. Typical operations for valuation of stochastic financial models need to be translated into the provided parallel operations. The functional approach we take enables us to provide guarantees of correctness, and – to some extent – performance estimations, for such translations. Sections 4.2 and 4.3 expand on previous and related work in the area of parallel functional programming and parallel hardware support.

**High-Performance Backends for Novel Hardware** Embracing novel parallel hardware like GPGPUs is an integral part of HIPERFIT. According to our holistic research approach, models and language framework will be designed with execution on next-generation processors in mind from the start, mapping the parallelism that is expressed by the functional programming activities onto a number of parallel computer architectures. In this research theme, activities will start by optimising existing algorithms and implementations, and profit from synergies with other scientific computing activities on parallel hardware. We follow a byte-code based approach and just-in-time compilation, and ultimately intend to deliver a full high-performance backend tailored for financial and scientific applications.

4 Functional Programming and HIPERFIT

4.1 Domain Specific Languages for Financial Applications

**Pervasive Trend to Domain Specific Languages** Declarative domain-specific languages (DSLs) capture knowledge of application experts in tailor-made constructs and thereby offer great programming comfort, operating at a high level
of (programming) abstraction. DSLs are so widespread and successful in practice that it is easy to overlook them: Logical data modeling and declarative querying, with high-level support for physical storage layout (particular index data structures) and automatic query optimization, as embodied in Relational Database Systems (RDBMSs); functional dependencies between atomic, vector- and matrix-based data, with automatic incremental recomputation, as embodied in spreadsheets; structural specification of strings, with automatic generation of provably efficient streaming processors, as embodied in regular expression (“lexing”) and context-free grammar (“parsing”) tools.

That said, programming language research has only recently discovered DSLs as a research area and capitalised on the notion [29]. Simultaneously with the rise of the term in research, one could observe DSL technology invading profitable commercial domains. For example, the Cryptol language [23] enables constructing reliable cryptographic software and hardware implementations with ease and high assurance. Recently, we also see some proposals for “DSLs” for parallel programming [38], or specifically for next-generation parallel hardware, GPGPUs or FPGAs. However, whether to really label these “DSL” is a debatable subject: A particular target platforms definitely does not constitute an application domain, and the particular field hardly exposes characteristics which would justify DSL development (special notation, automation, data structures [29]). These approaches are also usually not focusing particular application domains. We are not aware of many scientific projects combining a proper DSL approach with novel parallel hardware. Notable exception are a relatively new project Diderot [43] (a “parallel DSL” for image analysis), and the Feldspar project [13] which targets GPGPUs for high-performance signal processing using a DSL approach.

**DSLs in Finance** Financial applications have been identified as a promising DSL area relatively early. Researchers have successfully modeled and analyzed financial instruments [32], commercial contracts [2], and risk management [5] using DSL technology. The French company LexiFi, one of the industrial partners in HiPERFIT, has maturated the research on financial DSLs [32] into a language MLFi [24], which is embedded into ML as a combinator library for describing contracts and valuation (called a “domain-specific embedded language”, DSEL).

The hallmark feature of such contract languages is that they allow more complex instruments/contracts/risk models to be build up by composing simpler, often reusable, components that can be shared amongst different instruments. Also, the same domain-specific descriptions enables different interpretations. For instance, a description of financial instrument in MLFi can be used both for pricing the instrument and for scheduling the instruments (that is, managing when options and obligations described in the instrument are to be exercised. Based on the cornerstone MLFi, Lexifi today markets an integrated software solution for designing, pricing, analysing, and processing complex financial products. Their products prove useful for a broad customer community: structures, salespeople, quantitative analysts, risk managers and middle office professionals as well as in various fields: investment banks, asset managers and private banks.
Project Goal: DSL Framework for Finance  The general goal of DSLs is to support fast implementation, ease of change, extensibility, reusability across financial institutions, maintainability and low total cost of ownership (TCO) for the domain expert as a user. We want to create a framework for financial information applications which covers various applications: reporting to auditors and public authorities, data communication with clearing houses, internal reporting and statistics, computations for the purpose of internal risk management, and flexible integration for standard routines such as accounting and confirmation processing.

DSLs for financial instruments are today commonly used in many companies, but often mix contract and valuation aspects. A crucial goal of HIPERFIT is to design similarly expressive languages to describe the valuation process (using probabilistic computation and simulation), and to achieve clear separation and interfaces towards a universal valuation engine. We will investigate existing DSL approaches in the different areas and experiment with combining them to identify the lines of separation and useful language features. According to the overall span of HIPERFIT, we especially target novel parallel hardware. To our knowledge, this particular combination of financial application domain and high-performance hardware is unique.

4.2 Parallel Functional Programming in HIPERFIT

Why parallel functional programming matters... 4 The advantages of (parallel) functional programming have been stated often [19], so we just refresh the reader’s memory here: Functional programs are easy to read and understand, program construction and code reuse are simplified (glue), and program are transformed, optimised and formally reasoned about with ease. More specific to parallel computations, absence of side effects makes data dependencies and inherent parallelism obvious, (purely) functional parallel programs are deterministic irrespective of the evaluation order, and reduction semantics is inherently parallel, to start with. Last but not least, higher-order functions can nicely describe common parallelisation patterns as skeletons [11,34], without getting lost in technical details or particularities of the concrete algorithm. In all, irrespective of the concrete programming model, the high abstraction provided by functional languages makes them suitable languages to conceptually describe parallelism, in an executable specification.

Models, paradigms and classification  A number of programming models for parallel functional programming have been developed, which can be categorised along different aspects of programming and implementation. In the following, we attempt to give a concise overview supported by some examples (mainly) from the Haskell world, but necessarily not exhaustive. Interested readers can find other appropriate surveys [19,40].

4 In reverence to Hughes [20] and Hammond/Michaelson [19, Introduction].
A useful criterion for classification—of parallel programming models in general—is the degree of explicitness provided by the respective programming language or paradigm. Skillicorn and Talia [37] subdivide explicitness further along several aspects: decomposition, mapping, communication, and synchronisation, as increasing degrees of explicitness for parallel subcomputations. The main credo in functional languages being high abstraction, it is not surprising that most approaches to parallelism try to limit the programmer’s control of parallelism. Parallelism should ideally be non-invasive, i.e. not require large changes to a program’s source code. In the extreme, inherent parallelism exploited stems from the reduction semantics, for example in parallel Haskell (pH) [1]: lazy graph reduction is changed to eager evaluation for performance. However, experience has shown that such completely implicit approaches are of limited use. The predominant category is a mid-level of “controlled parallelism” [19], where programmers specify parallelism, while details are left to the language implementation. However, the understanding of explicitness varies, and especially the interesting mid-level remains vague and open to interpretation.

One approach is to indicate inherent parallelism in a functional program by annotations or special evaluation combinators, to inform compiler and runtime system about whether an independent computation should be done in parallel. This is the model of Glasgow parallel Haskell [42]. Another language in this spirit is concurrent Clean [33]; here, annotations have mandatory operational semantics. Data parallel languages also fall in this category. These languages use special bulk types and operations with parallel implementation. Examples are NESL [8] or the data parallel Haskell NEPAL [9] (and its newer variant [22]), and upcoming languages targeting GPGPUs [26,10].

A higher degree of execution control is achieved when the programmer explicitly specifies parallel scheduling. Programs with controlled parallelism are real parallel programs that expose their parallel behaviour. Examples include Hudak’s para-functional programming approach and successors [30]), or the evaluation strategies approach (as a high-level GpH [41] recently overhauled [28]), which enables to force evaluation of subexpressions to a certain degree (in parallel or sequentially). Skeleton-based parallelisation [12] could be ranged in this category since, commonly, the programmer has to explicitly choose the algorithmic pattern implemented by a certain skeleton, and to follow it. However, we prefer to categorise them as implicit (likewise Skillikorn/Talia [37]), since a skeleton’s parallel implementation is entirely hidden in a libraries.

Some functional languages go further and let the programmer completely control parallel execution, thereby augmenting the language expressiveness from transformational to concurrent, interactive and distributed systems. Often we find the concept of processes and channels between them to define process networks. The language Eden [25] is the major representative of this approach in the Haskell world. Eden retains a mostly [6] functional interface, with a notion of processes specified by their input-output mapping, and implicitly connected via channels which may transfer data as streams. The language Caliban [39] is an-
other example, but restricts parallelism to statically analysed process networks, while Eden process network can evolve dynamically during runtime.

Both Caliban and Eden are implicit about the communication details and synchronisation. Going even further, we find functional languages with *explicit message-passing and concurrency*. While these languages, for example Concurrent ML [35] and Concurrent Haskell [21] are primarily intended for distributed and interactive systems, they are also often used to achieve parallelism (speeding up a single computation). In fact, parallelism and concurrency are all too often mentioned together, leading to an endless debate on their relationship. Our position: (a) Concurrency can be an implementation tool for parallel algorithms, (b) functional languages allow for more deterministic models to implement parallelism, and (c) experience has shown that the large degree of control offered by concurrency abstractions and explicit message passing can prove useful for advanced parallel implementations [6].

**Project Goal: Tailored Parallel Functional Language** Within HIPERFIT, we aim to develop a functional language that can be productively used to express computations in mathematical finance, and which exposes inherent parallelism in these computations. Driven by the concrete application domain of financial modeling, we will identify common computation patterns and their potential for parallelisation. Potentially parallel computations should be easy to extract and transform into explicitly parallel operations on a variety of modern parallel platforms, possibly through an intermediate vectorised bytecode language. The paradigm of data parallelism [8] appears to be a good match for the HIPERFIT application domain: it enables concise and long-term maintainable specifications of a wide variety of inherently parallelisable computations, without committing to any particular implementation strategy or execution environment. Its purely functional semantics allows for a compositional cost model, and facilitates correctness proofs and performance estimates.

At a later stage, we expect the DSL development for financial models to yield additional or modified requirements. Useful abstractions and patterns of parallelism will be identified from working on concrete projects. Ultimately, our language should be extended with specific typical operations tailored to the application domain, risk analysis and valuation in a financial context.

### 4.3 Support for Multicore and Novel Parallel Hardware

In the previous section, we have motivated our functional approach by a number of historic achievements of relevance, based on more than 20 years of research in parallel functional programming. Yet, it is interesting to see how much the availability of advanced GPGPU hardware in practice changes the scientific landscape. GPGPUs are made for simple embarrassingly parallel problems with minor memory requirements. Parallel software has often been built as a match to existing well-performing and well-understood hardware. Functional approaches claim to capture parallelism at a more abstract level, but recent publications about
GPGPU programming in functional languages focus exactly on these simple embarrassingly parallel problems, where quick success can be expected.

Especially for accelerating financial simulations, the approach of modern GPGPUs appears promising; we already know that especially Monte-Carlo methods can get massive speedups, due to their simple, embarrassingly parallel nature. This holds not only for finance, but also for various scientific areas using Monte-Carlo simulations, for instance particle physics and computational geo-science. Today, we find several language bindings to GPGPU accelerators in the Haskell research community. They realise easy data parallelism on special parallel vectors (Nikola [26]) or arrays (Accelerate [10]). These research prototypes deliver important insight for future GPGPU language design and pragmatics, but we still have a way to go towards making this research software work in practice for the average programmer or domain expert. And as mentioned, we observe an antithetic trend in scientific computation: scientists of various disciplines choose to operate at the lowest abstraction level API, Cuda C code.

Before GPGPUs became the prodigy of parallelism, a first wave of interest for parallelism was induced by multicore. Having several cores available is a mere normality today, yet major functional languages have only recently optimised their support for shared-memory multicore platforms. The high-level of the languages, and implementation traditions, makes it sometimes very hard to optimise locality, but considerable results have been obtained [4,7,27], and even entire new projects for multicore were set up (for instance, Manticore [16]). With the movement towards OpenCL [31], both multicore processors, GPUs, and future heterogenous manycore architectures can be captured in a single computational idiom. OpenCL is supported by major manufacturers of novel hardware, and HIPERFIT will concentrate on advancing its development and use as an intermediate target language.

On another note, a recent effort worth mentioning is the initiative to make parallel Haskell apt for wide-spread commercial use, initiated by the Well-Typed consultancy and sponsors [3]. One of the first activities is to revive Haskell-MPI (from 2001) – this seems to be a major industrial demand, while leading researchers consider message passing “harmful” [18]).

So, is parallel Haskell taking off now?

Various activities are going on and in diverse directions, yet some promising research has not reach industrial maturity, and dedicated industrial support seems to go in a different direction.

We conclude that there is important work to do! Our intention with HIPERFIT in this angle is to contribute to existing approaches by ripening impulses from practicioners. We will closely follow latest trends in parallel functional programming. At the same time, we hope to contribute our own high-performance backend in the long term, which follows a bytecode approach and aims to be as general a backend for bulk data as we can realise.
5 Project Start and First Activities

Integrating Valuation and Contract Specification. The major use case of existing contract specification languages is valuation, to determine the value of a contract ahead of time (and based on uncertainties, stochastic variables). Existing contract languages have usually been developed together with a valuation semantics from the start. Based on a probabilistic model of unknown variables (for instance, modeling changes in interest rate for zero-coupon bonds), a range of possible outcomes and their probabilities is computed. A simple stochastic method for valuation is Monte-Carlo simulation, which includes inherent parallelism by nature. More advanced and expressive models might lead to a large number of possible outcomes and are thus computationally intensive, again parallelisation can hopefully lead to faster results.

![Figure 2. Integration Overview for Contracts, PFP, and Data Parallelism](image)

A recently started project in HIPERFIT prototypes a Haskell software which combines existing technologies for accelerated stochastic contract valuation. Figure 2 gives an overview of the evaluated technologies. As one strand of the activities, we evaluate existing GPGPU support in Haskell, namely the Nikola [26] DSEL and the accelerated Haskell array library [10], to offload computations on vectors to the GPGPU. Another aspect to integrate is the domain-specific approach to Probabilistic Functional Programming (PFP) [14]. This DSEL separates the stochastic method from the model and is thus helpful in structuring the implementation of our intended parallel valuation engine. Ultimately, we aim at a fully parametric valuation, where every valuation implementation \( V_M \) based on a model \( M \) can simply be written as \( V(M) \) where \( V \) is the valuation function parameterized over an arbitrary model.

Port of Data.Array.Accelerate to OpenCL. In view of our general goal to use and produce open standards in HIPERFIT\(^5\), we would like to pave a way

---

\(^5\) Software developed in HIPERFIT is jointly owned by all respective contributors, and University-produced software should be made available, as a matter of principle.
towards using the standard OpenCL [31] rather than the proprietary Cuda for GPGPU computations. We are therefore porting Data.Array.Accelerate [10] to OpenCL. The technology for this accelerated array library is well-understood, we expect to mainly solve technical and engineering hurdles here. As a by-product, a new library of OpenCL bindings will be created. At a later time, we might also be able to maturate the Nikola [26] research to better usability by non-experts, and port it to OpenCL as well.

As discussed earlier (see Section 4.3), the GPU platform and programming model appears to be tailored, if not even rigidly limited, to data parallelism. Control structures are very limited, memory accesses are entirely explicit, recursion is not possible, branching constructs execute both alternatives. On the other hand, precisely these properties could provide the magic wand for cost analysis and thereby performance prediction of parallelised valuation code. In view of this long-term goal, it is a strategic decision to generate know-how about GPU bindings involving embedded compilation in the HIPERFIT context.

Work for these two projects is underway at the moment, and we intend to present results and make the software available to general public in the future.

Other activities. Work has also started in other research areas of HIPERFIT, on projects which are less relevant to a functional community. To give a general idea of how our project-based working methodology looks like in practice, we mention some other activities. One interesting area for our partners is to parallelise random number generation (in a reproducible manner). A HIPERFIT project is investigating existing research to extract best practice on using GPGPUs for this problem. A second strand of activities concerns understanding existing software infrastructure of a typical bank, how best to utilise parallelism by cycle-scavenging, and which activities are the daily routine for trader support. In another project, we want to take the perspective of an informed economist on the topic of instrument valuation, by creating a survey and classification of financial instruments and models. Parallel implementations of selected valuation models will follow, which can be structured to reflect the generalities that have been identified. The implementation work also serves to evaluate another declarative parallel language (to be selected) and to identify recurring patterns and potentially useful features for later DSL development.

6 Conclusions

We have presented motivation, goals and methods of the HIPERFIT research center, a joint activity of researchers in mathematical finance, programming languages, parallel computing, and computer systems. In order to meet new and increasing computational needs of a complex global industry of major impact, HIPERFIT aims at integrated solutions which transcend a single researcher’s field of expertise, and explicitly fosters interdisciplinarity and practical relevance through its paradigm of project-driven research themes.
We want to develop advanced new methods in mathematical finance and work towards a framework of domain-specific languages to express financial instruments, models and valuation methods. Parallelisation techniques using a functional approach should both lead to efficient parallel execution on novel hardware, and leave the code accessible for proofs of semantic properties and, to some extent, performance predictions.

The goals of HIPERFIT which relate to programming languages appear to carry the highest risk, but promise the best long-term investment at the same time. Past research on parallelism concepts has often come to success and innovation by focusing on particular application domains. Immediate practical use and challenging problems derived from practice are a good touchstone for research. Especially because of the unique combination of advanced programming language technology and parallelism envisioned in HIPERFIT, we consider it an exciting chance to promote research in DSLs and parallel functional programming, and hope to make it one of its major showcases.

The HIPERFIT Website: http://www.hiperfit.dk or http://www.hiperf.it.

References


185
Abstract. General purpose computing architectures are evolving quickly to become many core and hierarchical: i.e. a core can communicate more quickly locally than globally. To be effective on such architectures programming models must be aware of the communications hierarchy. We propose 4 new architecture-aware primitives for GPH that exploit information about task size and aim to reduce communication for small tasks, preserve data locality, or to distribute large units of work. We investigate 3 common paradigms like data parallelism on hierarchical architectures with up to 224 cores. The results show that the new primitives consistently deliver better speedup and scalability than existing primitives together with dramatically reduced variability. At times speedup is improved by an order of magnitude. In addition we report a preliminary investigation and demonstration of architecture aware programming models that abstract over the new primitives. In particular we propose architecture aware evaluation strategies and skeletons.

1 Introduction

General purpose computing architectures are inexorably exploiting many cores, with the number of cores following Moore’s Law. Future architectures will inevitably have a hierarchical, or tree-like, communications structure. To exploit such architectures, programming models must be aware of the communication hierarchy.

Glasgow Parallel Haskell (GPH) [7, 4]) is a well-established, and widely used parallel extension to Haskell. Parallelism in the GPH model is achieved using only two primitives, \texttt{par} and \texttt{pseq}. Evaluation strategies are higher-order, polymorphic functions that provide high-level control of parallelism by abstracting over the primitives [8]. Currently the GPH programming model is oblivious to the underlying architecture: any unit of work may be executed by any core.

The paper makes the following research contributions:

– We propose 4 new architecture-aware primitives for GPH on architectures with hierarchical communication. The primitives exploit information about task size (thread granularity) and aim to preserve data locality or to distribute large units of work (Section 3).
- We investigate whether the new architecture aware primitives can deliver improved performance on a hierarchical architectures, considering simple examples of common paradigms: data parallel, divide-and-conquer, and nested parallelism. The evaluation is based on architectures up to 224 cores. (Section 4).
- We make a preliminary investigation into architecture aware programming models that abstract over the new primitives. In particular we propose architecture aware evaluation strategies, and architecture aware skeletons. Some of the new abstractions are used in the programs measured (Section 5).

2 Background

2.1 The Trend Towards Hierarchical Architectures

Physical limitations and manufacturing technologies are driving general purpose computing architectures inexorably towards many cores, with the number of cores following Moore’s Law. It is widely anticipated that future architectures will both evolve quickly, and have hierarchical communications structures. Already the most common parallel architectures are clusters of multicore nodes, with 3 levels in the hierarchy: a threads on the same core can communicate most quickly with a thread on the same core, more slowly with a thread on another core in the node, and slower still with threads on remote nodes. The communication hierarchy is likely to become deeper as the number of cores increase. For example the number of cores sharing the same memory is likely to be restricted, and hence many core architectures may introduce another level within a node.

Figure 1 illustrates both a virtual hierarchical architecture, and the specific instance that we use for the measurements later in the paper. The virtual architecture comprises a tree, possibly unbalanced, and where the degree of the
nodes may vary. We expect communication at lower levels in the tree to be faster than at higher levels. The measurement architecture networks a Beowulf cluster of 8-core multicore nodes with a freestanding 8-core multicore (lxpara3). The communication hierarchy has four levels, so from a Beowulf core Level 0 is on the same core; level 1 is to another node on the same core; level 2 is to a core on another Beowulf node; level 3 is to a core on the freestanding multicore (lxpara3).

To ground the measurements given in Section 4, the machines are located at Heriot-Watt University (MACS). lxpara3 is an eight-core 8GB RAM, HP XW6600 workstation comprising two Intel 5410 processors each running at 2.33GHz. The 32 Beowulf cluster nodes each comprise eight Intel 5506 cores running at 2.13GHz and 6GB RAM. All machines run Linux, i.e. CentOS 5.5. The Beowulf nodes are connected with Baystack 5510-48T switch with 48 10/100/1000 ports. Both Beowulf and lxpara3 are connected to the network with Extreme Networks Summit 400-48t, 48 10/100/1000BASE-T, 4 mini-GBIC, Extremeware

2.2 Glasgow Parallel Haskell (GpH)

Primitives Pure parallelism in Haskell is achieved using only two primitives, \texttt{par} and \texttt{pseq}, both with type \texttt{a -> b -> b}. The \texttt{par} combinator introduces a potential for parallel evaluation. When \texttt{par} is applied to two arguments, it returns the value of its second argument, while its first argument is possibly evaluated in parallel. We say “possibly”, because as far as semantics of the program is concerned, the result of \texttt{par a b} is always \texttt{b}; it makes no difference to the meaning of the program whether \texttt{a} is evaluated in parallel or not. We should think of \texttt{par} as an annotation; it merely hints to the Haskell implementation that it might be beneficial to evaluate the first argument in parallel.

In general, it is not enough to provide \texttt{par} alone, because when suggesting that something is to be evaluated in parallel, it is useful to be able to say what it is to be evaluated in parallel \emph{with}. Haskell neither specifies nor requires a particular order of evaluation, so normally the programmer has no control over this aspect of their program’s execution. In particular, the programmer has no control over when a particular call to \texttt{par} will be evaluated, or what will be evaluated before or after it (or indeed in parallel with it). This is the reason for \texttt{pseq}: a call \texttt{pseq a b} introduces an order-of-evaluation requirement that \texttt{a} be evaluated before \texttt{b}.

Evaluation Strategies Evaluation strategies [5], or “strategies” for short, are a key abstraction for adding pure, deterministic, parallelism to Haskell programs. Figure 2 shows the new version of Strategies [5] that use an Eval monad, and a Strategy on some type \texttt{a} is a function to \texttt{Eval a}. The \texttt{using} function applies a strategy, and \texttt{dot} composes strategies. \texttt{rpar} and \texttt{rseq} are strategies corresponding to \texttt{par} and \texttt{seq} respectively.

Parallel specifications can be built up in a compositional way, for example \texttt{evalList s} sequentially applies strategy \texttt{s} to every element of a list. Similarly \texttt{parList s} applies the strategy \texttt{s} to every element of the list in parallel. Moreover
data Eval a = Done a

instance Monad Eval where
  return x = Done x
  Done x >>= k = k x

runEval :: Eval a -> a
runEval (Done a) = a

type Strategy a = a -> Eval a

using :: a -> Strategy a -> a
x 'using' s = runEval (s x)

dot :: Strategy a -> Strategy a -> Strategy a
s2 'dot' s1 = s2 . runEval . s1

rseq :: Strategy a
rseq x = x 'pseq' return x

rpar :: Strategy a
rpar x = x 'par' return x

evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x; xs' <- evalList s xs; return (x’:xs’)

parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar 'dot' s)

parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap s f xs = map f xs 'using' parList s

Fig. 2. The Essence of the New Evaluation Strategies

the parallelism can be specified independently of the main computation, so for example the computational part of parMap is unchanged and isolated from the parallel control, a parList.

3 New Architecture-Aware Primitives

3.1 Virtual Architectures

For some problems, e.g. regular problems like many matrix manipulations, optimal performance can be obtained on a specific architecture by explicitly placing threads within the architecture. However, many problems don’t exhibit this regularity. Moreover explicit placement prevents performance portability: the program must be rewritten for a new architecture, a crucial deficiency in the presence of fast-evolving architectures.

To avoid these deficiencies we, like others [6], propose language constructs that expose a virtual architecture rather than the actual architecture. Clearly
the virtual architecture must be readily mapped to physical architectures. In addition our primitives minimise prescription: they identify sets of locations where the thread may be placed. Moreover we support performance portability by isolating the architecture-specific parts of the program in just a few functions that can be refactored for a new architecture.

3.2 Mapping Parallelism to an Architecture

Broadly speaking there are two challenges to be solved simultaneously when controlling parallelism on an hierarchical architecture, we must

Limit the communication costs for small computations. This entails limiting how far small computations are communicated, and requires information about thread granularity (i.e execution time). Without this information programs often have poor resource utilisation as the system is saturated with small threads [2]. Thread granularity information may be obtained from a number of sources, for example from some resource analysis, by profiling, by the programmer. We do not address the problem of obtaining this information here. Although the examples in the following section use granularity information from program parameters, we have adapted the runtime system to store granularity information with each spark if required.

Keep all cores busy, for example at system start up we must quickly distribute work to all cores. This entails sending large grain computations long distances over the communications hierarchy.

3.3 New Primitives

The new primitives are summarised in Figure 3. To limit the communication costs for small computations, or to preserve data locality, we propose parBound that behaves like par, except that it takes an additional integer parameter specifying the maximum distance in the communication hierarchy that the computation may be communicated. The distance represents a level in the hierarchy illustrated in Figure 1. So parBound 0 means the computation may not leave the core, parBound 1 may be communicated within the shared memory node, parBound 2 that may be communicated to another node in the beowulf cluster, and parBound 3 that it may be may be communicated freely to any core in the machine. parBound illustrates a key characteristic of our primitives,

parBound :: Int -> a -> b -> b
parAtLeast :: Int -> a -> b -> b
parDist :: Int -> Int -> a -> b -> b
parExact :: Int -> a -> b -> b

Fig. 3. New Architecture Aware Primitives

namely that while placing restrictions on how work is communicated, they aim for minimal prescription. That is, the primitives identify sets of locations where
a computation may be performed, and the runtime system is free to place the computation within this set. These sets often include multiple levels in the communication hierarchy. For example a computation sparked by a `parBound 1` may be placed on any core in the shared memory node, and a computation sparked by a `parBound 2` may be placed on any core in the set identified in Figure 1.

`parAtLeast` is a dual to `parBound`, as it takes an additional integer parameter specifying the the minimum distance in the communication hierarchy that the computation may be communicated. The idea is to communicate large-grain computations large distances, for example to flood a machine at the beginning of the execution. On the example architecture in Figure 1, `parAtLeast 3` means that the computation must be sent the greatest distance in the communication hierarchy, i.e. to `lxpara3`, `parAtLeast 2` that the computation must be communicated at least to another node in the beowulf cluster, `parAtLeast 1` that the computation must be communicated at least to another core within a shared memory node, and `parAtLeast 0` that it may be may be communicated freely to any core in the machine. `parBound` and `parAtLeast` can be generalised by the `parDist` primitive that specifies both a lower and upper bound on communication distance, as illustrated in Figure 4. Here `maxLevel` represents the maximum level of the communications hierarchy. As it is the most basic primitive, `parDist` is implemented in the GUM runtime system and creates sparks labelled with minimum and maximum communication distances.

`parDist` can also be used to define other primitives, for example `parExact` specifies an exact level, although not a specific processor. `parDist` can also be parameterised to capture other notions, for example `parDist n-1 n` specifies that a spark may be communicated to a core residing at either level `n-1` or level `n`, and `parDist n-2 n` is similar.

```haskell
parBound:: Int -> a -> b -> b
parBound n = parDist 0 n x y

parAtLeast:: Int -> a -> b -> b
parAtLeast n = parDist n maxLevel x y

parExact :: Int -> a -> b -> b
parExact n = parDist n n x y
```

Fig. 4. Architecture Aware Primitive Definitions

Table 1 summarises the new primitives in comparison with `par`. Thread creation remains optional for all primitives, likewise no specific location is identified by any of the primitives. The primitives may, or may not, restrict placement within the communication hierarchy, and `parExact` identifies a specific hierarchy level. Work distribution for all primitives is dynamic and passive, that is, idle cores seek work, and select only sparks for the appropriate communication level.
4 Architecture Aware Primitive Evaluation

We first investigate whether the new architecture aware primitives can deliver improved performance on a hierarchical architectures. We do so by considering common paradigms, namely data parallel, divide-and-conquer, and nested parallelism.

4.1 Divide-and-Conquer Parallelism

parFibDist :: Int -> Int -> Int
parFibDist 0 t = 1
parFibDist 1 t = 1
parFibDist n t
| n <= t = nFib n
| otherwise = parDist min max x (y 'pseq' (x + y + 1))
where
  x = parFibDist (n-1) t
  y = parFibDist (n-2) t
  (min, max) = findLevel n
nFib n = nFib (n-1) + nFib (n-2) + 1
findLevel ::(Ord a, Num a) => a -> (a, a)
findLevel x
| (x <= 46) = (0,0)
| ((x > 46) && (x <= 47)) = (1,1)
| ((x > 47) && (x <= 48)) = (2,2)
| otherwise = (3,3)

Fig. 5. parFibDist Program

To investigate the new architecture aware primitives for divide-and-conquer parallelism we use the parFibDist version of the parallel nfib function shown in Figure 5. Here \( n \) is the Fibonacci number \( t \) is the threshold value below which we use sequential computation (nFib). Above the threshold \( x \) is sparked with a parDist parameterised by levels computed by findLevel.

findLevel is a programming abstraction that is used, with the parameters specified in Table 2, for several experiments in this section. Table 2 shows possible configurations of findLevel function, ordered from left to right as increasingly specific spark placement.

Figure 6 compares the performance of par and the architecture aware primitives on the architecture specified in Section 2.1 configured with 1, 2, 4, 8 or 16 cores. The cores are distributed equally among machines. Throughout the results section each data point, unless otherwise stated, is the median of 3 executions.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>par</th>
<th>parBound</th>
<th>parAtLeast</th>
<th>parDist</th>
<th>parExact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Creation</td>
<td>Optional</td>
<td>Optional</td>
<td>Optional</td>
<td>Optional</td>
<td>Optional</td>
</tr>
<tr>
<td>Placement</td>
<td>Not Restricted</td>
<td>Not Restricted</td>
<td>Not Restricted</td>
<td>Not Restricted</td>
<td>Not Restricted</td>
</tr>
<tr>
<td>Hierarchy Placement</td>
<td>Not Restricted</td>
<td>Restricted</td>
<td>Restricted</td>
<td>Restricted</td>
<td>Specific</td>
</tr>
</tbody>
</table>

Table 1. GpH par Primitive Comparison (Increasingly Specific)
<table>
<thead>
<tr>
<th>Block Size</th>
<th>par</th>
<th>parBound</th>
<th>parAtLeast</th>
<th>parDist n-1 n</th>
<th>parDist n-2 n</th>
<th>parExact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Small</td>
<td>0 – maxLevel</td>
<td>0 – 0</td>
<td>0 – maxLevel</td>
<td>0 – 0</td>
<td>0 – 0</td>
<td>0 – 0</td>
</tr>
<tr>
<td>Small</td>
<td>0 – maxLevel</td>
<td>0 – 1</td>
<td>1 – maxLevel</td>
<td>0 – 1</td>
<td>0 – 1</td>
<td>1 – 1</td>
</tr>
<tr>
<td>Medium</td>
<td>0 – maxLevel</td>
<td>0 – 2</td>
<td>2 – maxLevel</td>
<td>1 – 2</td>
<td>0 – 2</td>
<td>2 – 2</td>
</tr>
<tr>
<td>Large</td>
<td>0 – maxLevel</td>
<td>0 – 3</td>
<td>3 – maxLevel</td>
<td>2 – 3</td>
<td>1 – 3</td>
<td>3 – 3</td>
</tr>
</tbody>
</table>

Table 2. findLevel Configuration

![Graph showing parfibDist Speedup Comparison](image)

The results show that the architecture aware primitives perform better in almost all cases. A rare exception occurs on 16 cores where the parBound speedup is 3.24, and par is 4.8, and occurs as all primitives are subject to scheduling accidents. However, this degree of influence depends specificity of the spark: the more specific the spark, the less influence of scheduling accidents. Interestingly, all primitives scale as the number of cores increases except parBound which performs slightly worse on 16 cores. The best speedup achieved by parExact: a speedup of 5.62 on 16 cores.

4.2 Data Parallelism

To investigate the new architecture aware primitives for data parallelism we use two programs. The intention for both programs is to generate data parallel tasks of random thread granularity. Both programs compute some function on every element of a list. The first program, parMapList in Figure 7, splits the list into sublists of random sizes, and the second program parMapIntervals in Figure 8, splits the interval into subintervals of random sizes, and the variation in task sizes for the programs is shown in Table 3. Allparam is nested parallel program will be presented in Section 4.3. The difference between the programs is that parMapList communicates the list, where the parMapIntervals communicates only the start and end points of the interval. Both programs compute the Euler sumTotient on each list interval or sublist.
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Smallest Spark Runtime (s)</th>
<th>Largest Spark Runtime (s)</th>
<th>Number of Sparks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allparam</td>
<td>0.43</td>
<td>2.90</td>
<td>3003</td>
</tr>
<tr>
<td>parMapIntervals</td>
<td>0.19</td>
<td>55.00</td>
<td>276</td>
</tr>
<tr>
<td>parMapList</td>
<td>.14</td>
<td>40.47</td>
<td>277</td>
</tr>
</tbody>
</table>

Table 3. Task Size and Irregularity

```haskell
dataListtop :: ([Int] -> a) -> Int -> Int -> Int -> a
dataListtop f lower upper t = let randomList = mkRandom t list = splitWithsize randomList [lower,lower+1..upper] result = sum (parMapLevel f findLevel list) in result
```

```haskell
sumTotient :: [Int] -> Int
sumTotient xs = sum (map euler xs )
```

```haskell
parMapLevel :: (Ord a) => (Num a) => (a -> b) -> [a] -> [b]
parMapLevel f _ [] = []
parMapLevel f _ (x:xs) = parDist min max fx (fxs 'pseq'( fx : fxs))
where
  fx = f x
  fxs = parMapLevel f xs
  (min,max) = findLevel x
```

```haskell
splitWithSize::[Int] -> [Int]-> [[Int]]
splitWithSize _ [] = []
splitWithSize (b:bs) xs = xss
where
  xss = (take b xs):splitWithSize bs (drop b xs)
```

Fig. 7. parMapList Program

```haskell
dataIntervaltop f lower upper seed = let randomList = mkRandom seed intervalList = splitIntervals (lower,upper) randomlist result= sum (parMapLevel f findLevel intervalList) in result
```

```haskell
splitIntervals ::(Ord a,Num a ) => (a,a) -> [a] -> [(a,a)]
splitIntervals (lower,upper) (b:bs)
| ((upper-b-1) <= lower) =[(lower,upper)]
| otherwise = ((upper-b),upper):splitIntervals (lower,(upper-b-1)) bs
```

```haskell
mkRandom seed = let g = mkStdGen 1601 cs :: [Int] cs = randoms g randomList = map (‘mod’ seed) $ cs in randomList
```

Fig. 8. parMapIntervals Program
For both programs the architecture aware primitives distribute the work depending on the size of the sublist or interval: small intervals are executed locally and large intervals are sent to be executed on remote core.

Both programs have similar performance on 64 cores, so we present only the results from \textit{parMapList} program. Figures 9 compare the absolute speedups of \textit{par} and the architecture aware primitives for both programs on the architecture with up to 64 cores. We make the following observations:

- All architecture aware primitives perform better than \textit{par} sometimes by factor of 2, e.g. on 64 cores.
- All of the architecture aware primitives scale: as the number of processors increase, the speedup increases. \textit{par} however does not scale beyond 16 cores. The reason is \textit{par} does not provide any restriction on thread placement thus; small thread can be executed remotely which does not cover the communication cost.
- The maximum speedup of 25.1 is obtained from \textit{parDist n-2 n}.
- While \textit{parExact} gives good performance, it is vulnerable to adverse scheduling because it identifies a specific level. Less prescriptive primitives like \textit{parDist n-2 n} can give better performance on large architectures, e.g. \textit{parMapIntervals} on 32 and 64 cores.
- In absolute terms \textit{parMapIntervals} has 20% better performance than \textit{parMapList} as it communicates only a pair of numbers, rather than a list segment.

Figures 10 and 11 show the absolute speedups of \textit{parMapIntervals} and \textit{parMapList} programs measured on 8, 16, 32, 64,\ldots, 224 cores. The absolute speedups are calculated with respect to the optimised sequential runtime.
**Fig. 10.** *parMapIntervals* Speedups (224 Cores)

**Fig. 11.** *parMapList* Speedups (224 Cores)
(2252.60s) for the larger problem size of 140000. The corresponding runtime curves can be found in [1], and we make the following observations:

- As before, the architecture aware primitives consistently perform better than \texttt{par}: by a factor of 2.5 (454.4/175.9) in the worst case on 224 cores, and 10.2 (454.4/44.7) is the best case.
- All of the architecture aware primitives scale, but for this input \texttt{par} however does not scale beyond 32 cores.
- In both graphs \texttt{parExact} scales more smoothly, but \texttt{parDist n-1 n} ultimately delivers the best performance.

```haskell
mapparfib n t [] = []
mapparfib n t xs = parMapLevel (parFibOneLevel t) findLevel randomList
  where
    randomList = take n ( filter (>35) xs )

parFibOneLevel :: Int -> Int -> Int
parFibOneLevel t n
  | n < t = nfib n
  | otherwise = parDist 0 1 x (y 'pseq' (x+y+1))
  where
    x = parFibOneLevel t (n-1)
    y = parFibOneLevel t (n-2)
```

\textbf{Fig. 12. Allparam Program}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{speedup_comparison.png}
\caption{Allparam Speedups (64 Cores)}
\end{figure}

### 4.3 Nested Parallelism

To investigate the new architecture aware primitives for nested parallelism we use the \textit{Allparam} program in Figure 12 with top level data parallelism and nested
divide-and-conquer parallelism. More specifically it maps the parallel divide and conquer fibonacci function \texttt{parFibOneLevel} over a list of random integers in parallel using \texttt{parMapLevel}.

Figures 13 and 14 show the absolute speedups obtained with the primitives. The results are similar to those observed with the divide and conquer and data parallel programs: the architecture aware primitives consistently outperform \texttt{par}, with an exception at 224 cores. \texttt{parExact} and \texttt{parDist n-1 n} deliver the best performance. None of the primitives scales beyond 128 cores, and we attribute this to the task sizes being too small for large architectures, as illustrated by the maximum task size in Table 3.

4.4 Variability

Finally we investigate the performance variability induced by each primitive. The variability reflects how susceptible the work distribution policy of each primitive is to scheduling accidents. Table 4 shows the wide variation in runtimes for 11 executions of the three programs with the different primitives. We make following observation, taking \texttt{parMapIntervals} as an example for discussion:

- The architecture aware primitives have far less variation than \texttt{par}, with a range of 121s and a standard deviation (sd.) of 33s.
- \texttt{parDist n-1 n} and \texttt{parDist n-2 n} have the least variation (range 13.7s and 10.2s, sd.s 4.3s and 3.2s). It seems that having multiple levels available enables the runtime system to ameliorate scheduling accidents.
- Of the architecture aware primitives, \texttt{parAtLeast} has the worst performance.
### 4.5 Discussion

The results obtained for the prospective architecture aware primitives are consistent for all three classes of program investigated. Every architecture aware primitive consistently deliver better speedup and scalability than `par` with dramatically reduced variability. The `parExact` and `parDist n-1 n` primitives deliver the best performance, and for these programs with simple coordination `parExact` scales more smoothly. `parDist` is clearly the most powerful primitive, but requires the programmer to specify the most information, i.e. both an upper and lower bound.

In summary we recommend using both `parDist` and `parExact` primitive: `parDist` for expressive power and `parExact` for simplicity and performance.

### 5 Architecture Aware Programming

The functional programming philosophy is to construct high level, often higher order, programming constructs by abstracting over the primitives. So what are appropriate high level architecture-aware programming constructs? We have started to investigate both architecture aware skeletons and evaluation strategies.

The `findLevel` function in Figure 5 is already a useful abstraction. It computes upper and lower communication bounds from an argument.

#### 5.1 Architecture Aware Strategies

Having selected architecture aware primitives, we must now develop evaluation strategies using them. Defining a strategic version of the new primitives is straightforward, e.g. `rparDist min max x = parDist min max x (return x)`, and `parDistList :: Int -> Int -> Strategy a -> Strategy [a]` is analogous to `parList`. However, care must be taken when refactoring the strategies module both to preserve the existing strategies, and to make the higher order strategies parametric in the new primitives.
We use the new strategies to define some of the skeletons in the following section, and in some programs. For example the widely used Queen placement benchmark uses \texttt{pparDistList} to deliver speedup of 2.5 on four cores.

5.2 Architecture Aware Skeletons

Algorithmic skeletons capture common patterns of parallelism and are widely used, e.g. [3]. We have adapted some skeletons to use architecture aware primitives and strategies.

An Architecture Aware Parallel Map Skeleton The \texttt{parMapList} program in Figure 7 has already illustrated one such skeleton, namely \texttt{parMapLevel}. \texttt{parMapLevel} is a parallel map that sparks each list element with \texttt{parDist} parameterised with the levels identified by \texttt{findLevel}.

A Divide-and-conquer Skeleton An architecture aware version of a common divide-and-conquer skeleton is shown in Figure 15. \texttt{findLevels} added as a new parameter to exploit the communication architecture. The skeleton delivers improved performance compared with the original skeleton using \texttt{par} for both a sumEuler and a Fibonacci program. On 16 cores the maximum speedups are 7.6 and 7.0 for the architecture-aware sumEuler and Fibonacci and 2.2 and 7.5 for the original.

\begin{verbatim}
divConq :: (Ord a,Num a) => (a -> b) -> a -> (a -> Bool) ->
         (b -> b -> b) -> (a -> Bool) -> (a -> (Int,Int)) -> (a -> (a,a)) -> b

| not (divisible arg) = f arg
| otherwise = conquer l r
where
  (lt,rt) = divide arg
  (min ,max) = findLevel arg
  left = divConq f lt threshold conquer divisible findLevel divide
  right = divConq f rt threshold conquer divisible findLevel divide
  (l,r) = (left, right) 'using' strat
strat (l,r)
  | (threshold arg) = (evalTuple2 rseq rseq) $(l,r)
  | otherwise = (evalTuple2 (rparDist min max 'dot' rseq)
                (rparDist min max 'dot' rseq)) $(l,r)

Fig. 15. An Architecture Aware Divide-and-Conquer Skeleton
\end{verbatim}

6 Conclusion

In response to architecture trends towards hierarchical communications topologies, and rapid evolution of these topologies we propose to develop an architecture aware programming model for parallel Haskell. We start by defining 4 new architecture-aware primitives for GPH. The primitives exploit information about task size and aim to reduce communication for small tasks, preserve data locality, or to distribute large units of work. They do so by constraining communication abstractly and with as little specific prescription as possible, that
is the primitives identify layers of the communication hierarchy, and allow the implementation to dynamically control placement within the layer (Section 3).

We have evaluated the primitives using simple data parallel, divide-and-conquer, and nested parallel programs, each with irregular thread granularity. The evaluation shows that every architecture aware primitive consistently delivers better speedups, better scalability and far less variation in execution time than the existing par primitive. Because of that par does not provide any restriction on thread placement thus; small thread can be executed remotely. At times speedup is improved by an order of magnitude. Of the proposed primitives, parExact and parDist n-1 n deliver the best performance, parExact scales most smoothly, and parDist is the most expressive primitive. We recommend parDist for expressive power and parExact for simplicity and performance (Section 4).

We have started to develop high level architecture-aware programming constructs that abstract over the new primitives. In particular we are developing architecture aware evaluation strategies, and architecture aware skeletons. Some of the new abstractions, like findLevel and parMapLevel are exploited in the programs measured (Section 5).

In ongoing work we are further developing architecture aware evaluation strategies. We are using these strategies to construct and measure architecture aware versions of a suite of larger benchmarks.

References

Refactoring Parallel Haskell Programs

Christopher Brown¹, Hans-Wolfgang Loidl² and Kevin Hammond¹

¹ School of Computer Science, University of St. Andrews, UK.
{chrisb,kh}@cs.st-andrews.ac.uk
² School of Mathematical and Computer Sciences, Heriot-Watt University, UK.
hwloidl@macs.hw.ac.uk

Abstract. Refactoring Haskell programs —the process of changing the
structure of a Haskell program whilst preserving the program’s semantics—
has, hitherto, been applied to sequential Haskell 98 programs, identifying
well-understood patterns of abstraction. This refactoring process aids in
helping the programmer’s understanding of a program: increasing code
quality, programming productivity and code reuse.

In this paper we present a radically different approach to applying refac-
toring patterns to Haskell 98 programs: namely refactorings that aid
programmers in developing parallel Haskell programs, a notoriously dif-
ficult task for programmers who are unfamiliar with parallel concepts.
Specifically, we introduce a number of new refactorings that introduce
common parallel abstractions, such as divide and conquer using the latest
Strategies module and the infrastructure of HaRe, the Haskell Refactorer.
These parallel refactoring patterns target GpH: the common platform
for parallel computation in Haskell. We present the new parallel refac-
torings and then motivate their use with a worked example. With our
radical approach of applying software refactoring techniques to parallel
programming, we hope to allow programmers to truly starting thinking
parallel.

1 Introduction

Glasgow parallel Haskell (GpH) provides a model of semi-explicit parallelism
where the programmer only has to annotate expressions that could be executed
in parallel, by using the par primitive. Adding such primitives does not modify
the semantics of the program, which is a big benefit for program development.
Another benefit of this programming model is that all coordination and synchro-
nisation is managed automatically by the runtime-environment.

While this semi-explicit model of parallelism is very flexible and secure in
terms of the results that are generated, it is often difficult for new users to
identify useful sources of parallelism in a bigger application. This choice often
requires experience in parallel programming in GpH as well as information on
the computational costs of sub-expressions.

In this paper we use a refactoring approach for parallel programming. We
identify a set of small program transformations that can be used to introduce and
tune parallelism in a larger Haskell application. This process is computer-aided
but deliberately not automated: purely automated transformation systems could result in programs that the user no longer understands; this is precisely the opposite of what we are trying to achieve. Expert knowledge is still required to guide the system in introducing parallelism. The system itself helps the programmer by performing the routine tasks of modifying the program structure, for example by renaming the appropriate sub-computations or abstracting constants out of functions. It can also give warnings about ill-defined program fragments, such as unspecified evaluation degree.

1.1 Refactoring

When writing a first version of a program, often programmers plough ahead without paying any attention to design principles or programming style [2]. This often results in the programmer realising that a different, more considered, approach would have been much better; or that the context or specification of the problem has changed. Refactoring tools aim to solve this issue by providing tool support to modify the design of the program without changing its functionality.

The term ‘refactoring’ was first introduced by Opdyke in his PhD thesis in 1992 [3] and the concept goes at least as far back as the fold/unfold system proposed by Burstall and Darlington in 1977 [4], although, arguably, the fold-unfold system was more about algorithm change than structural changes. A key aspect of refactoring — illustrated by the ‘rename function’ operation — is that its effect is across a code base, rather than being focussed on a single definition: renaming a function will have an effect on all the modules that call that function, for instance.

The refactorings presented in this paper fall generally into two categories:

- **Parallelisation**: adding explicit parallel annotations to the code, or modifying an existing evaluation strategy.
- **Tuning**: small structural changes that improve a parallel algorithm, changing an evaluation degree or adding a threshold, for instance.

The refactorings presented in this paper are implemented in the Haskell Refactorer, HaRe [5]. A Haskell 98 refactoring tool developed at the University of Kent, HaRe provides refactoring support for the full Haskell 98 standard, and is built into the Emacs and Vi editor environments. HaRe currently provides a substantial number of structural and data-type based refactorings, aimed at refactoring pure sequential Haskell 98 programs. In addition, HaRe is built upon the Programatica [6] front-end for parsing, and the Strafunski library [7] for generic tree traversals. The combination of which is abstracted into a low-level refactoring API [8] for designing and developing refactorings. This API provides the user with an abstract syntax tree (AST) for the source program together with utility functions (tree traversal and tree transformations) to assist in the implementation of refactorings. HaRe is itself written in Haskell, and the refactorings it supports can be applied to both single and multi module projects.
1.2 Contributions

The main contribution of this paper is a worked example that demonstrates how parallelism can be systematically introduced into a sequential Haskell program that uses a top-level divide-and-conquer structure. We describe a sequence of simple transformations and elaborate how the system helps an inexperienced parallel programmer in producing parallel code, avoiding common pitfalls in the program design.

The parallelisation builds on the latest implementation of evaluation strategies [1], which provide a high-level notation for specifying parallelism, evaluation degree and evaluation order in a lazy functional language. Compared to the direct use of par annotations, they provide a more structured approach for introducing parallelism and therefore help the programmer in dealing with the complexity of large-scale parallel applications.

The contributions presented in this paper are:

- A number of new parallel based refactorings for GpH programs. Specifically, we concentrate on refactorings that introduce a divide and conquer parallel model.
- A refactoring case study, demonstrating the aforementioned parallel refactorings on a worked example. In particular, we demonstrate how the new parallel refactorings can help the programmer introduce parallelism.

2 GpH and Evaluation Strategies

To specify parallel execution, we use Glasgow parallel Haskell (GpH), a conservative extension of the purely functional, non-strict programming language Haskell. Two basic primitives generate pure parallelism and controlling evaluation:

\[
\text{par} :: a \rightarrow b \rightarrow b \\
\text{pseq} :: a \rightarrow b \rightarrow b
\]

Here, \text{par} is used to indicate a potential parallel evaluation. When \text{par} is applied to two arguments, the second is returned while the first is possibly evaluated in parallel. The use of \text{par} is merely an annotation, however: it gives no guarantee as to whether the first argument is evaluated in parallel or not, as this makes no difference to the meaning of the program. Expressing parallelism is not enough in Haskell, we also need an operator to specify evaluation order. Here \text{pseq} a b requires that a is evaluated before b.

Evaluation strategies [9,1] are a key abstraction for developing parallel programs in a compositional way, separating the coordination aspects of the execution from the computation aspects by providing an abstraction layer around \text{par} and \text{pseq}. To this end, an evaluation order monad Eval is introduced. This monad can be used to define parallelism, degree of evaluation and order of evaluation for specific program expressions. An evaluation strategy is then simply
a function over an expression, defining one such monad. It is applied to an expression, by executing the `runEval` operation of the monad. This uses familiar, monadic notation, and a rich repertoire of abstraction, to specify parallelism. In comparison to our previous definition of evaluation strategies [9] this formulation has improved modularity, which is demonstrated in a companion paper discussing variants of the above `par` primitives [10].

```haskell
type Strategy a = a -> Eval a
data Eval x = Done x

runEval :: Eval a -> a
runEval (Done a) = a

instance Monad Eval where
    return x = Done x
    Done x >>= k = k x
```

The following basic strategies are available: `r0` does not perform any evaluation at all, `rseq` performs the default evaluation of an expression to weak head normal form, `rdeepseq` performs a full evaluation of a data structure, and `rpar` generates (potential) parallelism for its argument. Note that `rdeepseq` requires an instance of the class `NFData`, which defines how to fully evaluate the data structure.

```haskell
r0 :: Strategy a
r0 x = return x

rseq :: Strategy a
rseq x = x `pseq` return x

rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x

rpar :: Strategy a
rpar x = x `par` return x
```

The following two functions are used to apply a strategy to an expression (`using`) and to compose strategies (`dot`), respectively:

```haskell
using :: a -> Strategy a -> a
x `using` s = runEval (s x)

dot :: Strategy a -> Strategy a -> Strategy a
s2 `dot` s1 = s2 . runEval . s1
```

As an example of composing a more sophisticated strategy, we define `evalList`, which operates on a list and is parameterised by the strategy that should be applied to the elements of the list. Based on this definition we can then define an instance that evaluates all elements in parallel, `parList`, by composing the
strategy parameter \( s \) with the parallel strategy \( \text{rpar} \). The possibility to compose strategies like this achieves a high degree of compositionality of this abstraction mechanism.

\[
\text{evalList} :: \text{Strategy } a \rightarrow \text{Strategy } [a] \\
\text{evalList } s \; [] \; = \; \text{return } \; [] \\
\text{evalList } s \; (x:xs) \; = \; \text{do } x' \leftarrow s \; x \\
\hspace{1cm} xs' \leftarrow \text{evalList } s \; xs \\
\hspace{1cm} \text{return } (x':xs')
\]

\[
\text{parList} :: \text{Strategy } a \rightarrow \text{Strategy } [a] \\
\text{parList } s \; = \; \text{evalList } (\text{rpar} \; \text{‘dot’} \; s)
\]

As an example of using evaluation strategies, the following function implements a parallel version of the well-known \( \text{nfib} \) function, generating parallelism for both recursive calls:

\[
\text{pfib } n \\
| \; n \leq 1 \; = \; 1 \\
| \; \text{otherwise } = \; \text{runEval } \; $ \; \text{do} \\
\hspace{1cm} x \leftarrow \text{rpar } (\text{pfib } (n-1)) \\
\hspace{1cm} y \leftarrow \text{rpar } (\text{pfib } (n-2)) \\
\hspace{1cm} \text{return } (x + y + 1)
\]

It is important to note that in the computation of the overall result, the values returned by the strategies \( \text{rpar} \) and \( \text{rseq} \) are used. This ensures that the parallelism generated for the recursive calls won’t be discarded. One slight disadvantage of this approach is that computational code is embedded in the strategy, in this case by returning \( x+y+1 \). While this can be avoided in specific cases, we want to separate computation and coordination as much as possible. To this end, we prefer a version that only returns the results of strategy applications in a tuple and uses them in composing the overall result outside the strategy.

\[
\text{pfib } n \\
| \; n \leq 1 \; = \; 1 \\
| \; \text{otherwise } = \; x + y + 1 \\
\hspace{1cm} \text{where } (x,y) = \; \text{runEval } \; $ \; \text{do} \\
\hspace{2cm} x \leftarrow \text{rpar } (\text{pfib } (n-1)) \\
\hspace{2cm} y \leftarrow \text{rpar } (\text{pfib } (n-2)) \\
\hspace{2cm} \text{return } (x, y)
\]

3 Parallel Refactorings

This section describes some new parallel refactorings that we have defined and implemented in HaRe. In particular, the following refactorings are described in this section: introduce parallelism (Section 3.1) and introduce thresholding (Section 3.2).
Before:

\[
\text{qsort} \ [] = [] \\
\text{qsort} (x:xs) = \text{left} ++ [x] ++ \text{right} \\
\text{where} \\
\text{left} = \text{qsort} (\text{filter} (<) x xs) \\
\text{right} = \text{qsort} (\text{filter} (\geq) x xs)
\]

After:

\[
\text{qsort} [] = [] \\
\text{qsort} (x:xs) = \text{left'} ++ [x] ++ \text{right'} \\
\text{where} \\
\text{left} = \text{qsort} (\text{filter} (<) x xs) \\
\text{right} = \text{qsort} (\text{filter} (\geq) x xs) \\
(\text{left'}, \text{right'}) = \text{runEval} $ do \\
\text{left'} <- \text{rpar left} \\
\text{right'} <- \text{rpar right} \\
\text{return} (\text{left'}, \text{right'})
\]

**Fig. 1.** Introducing *divide and conquer* parallelism via the *runEval* monad for quicksort

The main goal of these refactorings is to introduce and tune the parallelism in a program. Therefore, the main refactoring is to introduce parallelism. The other refactorings can be used to improve parallel performance, by increasing the granularity of the parallelism (introduce thresholding) and by modifying the order or degree of evaluation (modify evaluation degree). In some cases it is also useful to modify the structure of the code beforehand, enabling a refactoring that introduces or improves parallelism.

### 3.1 Introduce Parallelism

**Worked Example**  This refactoring is in two parts; firstly, the user selects a sub-expression, *e* (or binding) that they want to spark as a parallel computation. The refactoring then introduces an *Eval* monad defined locally to the sub-expression in a *where* clause, sparking the identified sub-expression in parallel. The sparked computation is then returned as a result of the monad, substituting uses of *e* with the sparked sub-expression, instead. Secondly, the user may add additional expressions to the *Eval* monad to be sparked in parallel with the original sub-expression.

As an example of introducing parallelism into a program, consider the *quicksort* example in Figure 1. In the example, the code on the left is the sequential quicksort version, and on the right we have a parallel version, using a parallel divide-and-conquer pattern. In order to perform this refactoring, we first select *left* within the *where* clause of *qsort* and select the *Introduce Parallelism* refactoring from the HaRe drop-down menu. HaRe then automatically adds a new pattern match definition within the *where* clause of *qsort*, binding a new name *left’* to a sparked version of the original *left*:

\[
\text{qsort} (x:xs) = \text{left'} ++ [x] ++ \text{right} \\
\text{where} \\
\text{left} = ... \\
\text{right} = ... \\
\text{left'} = \text{runEval} $ do \\
\text{left'} <- \text{rpar left}
\]

209
We note that rpar generates potential parallelism for evaluating left, using the
default evaluation degree. In addition, all calls of left within the right hand
side of qsort are replaced with calls to left'.

The next step is to add right to the Eval monad as an additional rpar
binding. To perform this operation, we first select the left' pattern within the
where clause and choose Active Eval Monad from the HaRe menu; this indicates
that we intend to modify this computation by adding further bindings to it. To
add another binding we then select right within the where clause of qsort and
select the Introduce Parallelism refactoring. HaRe then modifies the monad,
adding a new binding for the sparked right computation.

qsort (x:xs) = left' ++ [x] ++ right'
where
left = ...
right = ...
(left ', right') = runEval $ do
  left' <- rpar left
  right' <- rpar right
  return (left', right')

The right hand side of qsort is also modified by the refactoring so that all calls
to right are substituted for calls to right' instead.

At both steps of this refactoring, HaRe warns the user that the rpar anno-
tation uses a default weak head normal form evaluation strategy. This is very
useful, because inexperienced users of GpH often fail to enforce a deeper evalu-
ation degree on data structures that should be processed in parallel, thus pro-
ducing code containing a smaller amount of parallelism than expected. Further
refactorings to appear in the full version of the paper will allow the user to eas-
ily modify the evaluation strategy if they so desire. In practice, such decisions
should be guided by the costs of the computation, inferred either statically or
through profiling. In this paper, however, we do not discuss these aspects in
more detail and focus on the parallel refactorings, instead.

Introduce Parallelism The goal of this refactoring is to identify an expression
in the code that could be evaluated in parallel. This is done by introducing an
evaluation monad, or adding a new binding to an existing evaluation monad.

Add Parallelism - Transformation Rules

- Suppose \( f = e[x] \) where \( x \) is declared in \( f \) and free in \( e \). Upon selecting \( x \),
  the refactoring produces \( (e[x \rightarrow x']) \) stands for substituting \( x' \) for \( x \) in \( e \)):

  \[
  f = e[x \rightarrow x'] \text{ where } x' = \text{runEval } \& \text{ do } x' \leftarrow \text{rpar } x; \text{ return } x'
  \]

- If a where or let clause already exists in \( f \), \( x' \) is added to the clause that
  declares \( x \).
- x’ is a unique name in the scope in which it is defined. The name is generated by HaRe automatically, and can further undergo a renaming [11] refactoring, if required.
- All occurrences of x in the body of f are substituted for x’ in the scope in which x is defined.
- x must be a pattern binding declared in the scope of f. If x is a function then it cannot be used in the runEval and the refactoring will fail with an error message.
- If an Eval monad has been activated (see below), after selecting x to be bound within the activated monad, the refactoring produces:

\[
f = e[x -> x']
\]

where \((a',b',...,x') = \text{runEval} \$ do\]

\[
a' \leftarrow \text{rpar} \ a;
b' \leftarrow \text{rpar} \ b; \ldots; \ x' \leftarrow \text{rpar} \ x;
\]

\[
\text{return} \ (a', b', \ldots, x')
\]

Here a and b are defined in the scope of f. In addition, the activated Eval monad is modified to return a N-ary tuple (if it doesn’t already). Each tuple element is bound within the Eval monad. If the monad already returns a tuple, then the new binding (x’ in the example above) is added to the tuple and the pattern binding on the right hand side is extended to take x’ into account. If the monad only returns one binding, then it is modified to return a tuple: the first element being the original binding and the second being the newly introduced x’ binding. All occurrences of x are substituted with x’ in the scope in which it is defined.

Global Transformation Rules

- If x is a top-level pattern binding, then x’ is also added to the top-level scope.
- A new import declaration import Control.Parallel.Strategies is added to the list of import declarations in the module declaring f. However, if rpar or runEval are explicitly included in the list of identifiers to be hidden from the import of Control.Parallel.Strategies, then the refactoring will fail with an error message.
- If the module declaring f already declares import Control.Parallel.Strategies with an explicit list of imported identifiers, then rpar and runEval are added to the explicit import list, unless they are hidden.
- If import Control.Parallel.Strategies is qualified as, say, S then the introduced rpar and runEval are replaced with S.rpar and S.runEval.
- Finally, x must not be bound within a list comprehension or a monad, otherwise the refactoring fails with an error message.

3.2 Introduce Thresholding

Once parallelism is introduced to a program, the user often needs to tune the parallel performance to get good results. In this section we discuss one common
refactoring that is often used to increase the granularity of the parallelism, i.e. to generate larger units of parallelism.

More specifically, introduce thresholding allows the user to control the parallelism by disabling it if a selected value is below a threshold limit. The threshold limit is supplied as a parameter to the refactoring, and is used to generalise the threshold: a new argument is added to the function and all calls pass the threshold value in. This threshold can then be changed depending on where and how the function is called in the program.

For an example of Introducing a Threshold consider Figure 2. In the left of the figure, we have defined a version of fibonacci that uses a parallel divide-and-conquer pattern (introduced using the Introduce Parallelism refactorings as described in Section 3.1). Sometimes it is essential to turn off parallelism once the granularity of the computation becomes too small to warrant the generation of a separate thread. In order to add this threshold, we simply select \( n \) in the argument list to \( \text{pfib} \) and choose the Introduce Threshold refactoring from HaRe. HaRe then prompts us for a value for our threshold. In this example we enter a constant 20 as threshold value. In general, this can be an expression over the arguments of the function. The result of the transformed code is shown in the right column of the figure.

Specifically, the refactoring has added a new guard clause to the \( \text{runEval} \) monad: this guard clause is used for the parallel version of the algorithm. Note that a sequential (default) strategy is applied if \( n \) is not greater than the threshold value. Furthermore, \( \text{thres} \) has been added as an argument to \( \text{pfib} \). All calls to \( \text{pfib} \) — in the scope in which it is defined — are replaced with calls to \( \text{pfib} 20 \). Not only does this allow for the threshold value to be generalised for \( \text{pfib} \), it is also added to the beginning of the argument list in case \( \text{pfib} \) is partially applied somewhere: it is far easier to simply replace \( \ldots\text{pfib}\ldots \) with \( \ldots(\text{pfib} \ 20)\ldots \) in a partial application than doing extra work determining where, in the application of \( \text{pfib} \) the argument 20 must be included. This behaviour is designed to be consistent with the Introduce an Argument and Generalise Definition refactorings already defined in HaRe [11]

**Transformation Rules**

- Suppose
  
  \[
  f = e
  \]

  where \( n = \ldots \)

  \[
  y = \text{runEval} \ $ do \ e1
  \]

  then if we have previously cached \( y \), selected \( n \) and entered 42 at the HaRe prompt, the refactoring produces the following:

  \[
  f \ \text{threshold} = e
  \]

  where \( n = \ldots \)

  \[
  y \\
  | n > \text{threshold} = \text{runEval} \ $ do \ e1 \\
  | \text{otherwise} = \text{runEval} \ $ do \ e1[\text{rpar} \to \text{rseq}]
  \]
Fig. 2. Introducing a threshold to turn parallelism off when granularity is too small

- threshold is added as an argument to \( f \). The name is guaranteed by the refactoring to be unique to \( f \) to avoid capture.
- If the user enters 20 as the threshold value, all calls to \( f \) in the scope in which it is defined are replaced with calls to \( f \ 20 \).

4 Refactoring Case Study

In this section we present a simple but realistic example from the parallel section of GHC’s nofib suite. It illustrates how the refactorings presented in this paper can be used effectively to gain and tune a parallel implementation of a sequential divide-and-conquer algorithm. For our example we take a sequential Fourier Transform algorithm.

The initial sequential version of the algorithm is defined in Haskell as follows:

```haskell
ffth :: [ComplexF] -> [ComplexF] -> [ComplexF]
ffth xs us
| n > 1 = (ffmups (replikate fftEvn) (ffmus us (replikate fftOdd)))
| n = 1 = xs
where
  rvv = ffmus us (replikate fftOdd)
  rvv1 = replikate fftEvn
  fftEvn = ffth (evns xs) uEvns
  fftOdd = ffth (odds xs) uEvns
  uEvns = evns us
  evns = everyNth 2
  odds = everyNth 2 . tail
  n = length xs
```

The refactoring is as follows:

```haskell
pfib thres n
| n <= 1 = 1
| otherwise = n1' + n2' + 1
where
  n1 = pfib (n-1)
  n2 = pfib (n-2)
  (n1', n2') = runEval $ do
    n1' <- rpar n1
    n2' <- rpar n2
    return (n1', n2')
```

The modified version of the algorithm is defined in Haskell as follows:

```haskell
pfib n
| n <= 1 = 1
| otherwise = n1' + n2' + 1
where
  n1 = pfib (n-1)
  n2 = pfib (n-2)
  (n1',n2') = runEval $ do
    n1' <- rpar n1
    n2' <- rpar n2
    return (n1', n2')
```

The threshold is added as an argument to \( f \). The name is guaranteed by the refactoring to be unique to \( f \) to avoid capture.

If the user enters 20 as the threshold value, all calls to \( f \) in the scope in which it is defined are replaced with calls to \( f \ 20 \).
4.1 Stage 1: Introduce Parallelism

The first stage is to identify which components in \texttt{ffth} we would like to parallelise. Using the standard \textit{divide and conquer} approach, we would like to separate out the recursive calls to \texttt{ffth} in the bindings of \texttt{fftevn} and \texttt{fftodd}, so that \texttt{fftevn} is sparked in parallel to the evaluation of \texttt{fftodd}. In order to do this, we would like to use the \texttt{Eval} monad, defined in the strategies module. The first step is to therefore select the entity \texttt{fftevn} and choose the \textit{Introduce Parallelism} refactoring from the HaRe drop down menu. The refactoring then introduces a new pattern match in the \texttt{where} clause of \texttt{ffth}:

\begin{verbatim}
ffth :: [ComplexF] -> [ComplexF] -> [ComplexF]
fifth xs us
| n>1 = (ffmups (replikate fftEvn') (ffmus us (replikate fftOdd')))
| n==1 = xs
where
    rvv = ffmus us (replikate fftOdd)
    rvv1 = replikate fftEvn
    fftEvn = fifth (evns xs) uEvns
    fftOdd = fifth (odds xs) uEvns
    uEvns = evns us
    evns = everyNth 2
    odds = everyNth 2 . tail
    n = length xs

    fftEvn' = runEval $ do
      fftEvn' <- rpar fftEvn
      return fftEvn'
\end{verbatim}

It is important to observe that in this step the refactoring also substitutes occurrences of \texttt{fftEvn} within the body of \texttt{ffth} so that it uses the new \texttt{fftEvn'} binding instead.

The next step is to add \texttt{fftOdd} as a binding within the \texttt{fftEvn'} monad. The refactoring \textit{Add Binding to runEval} will let us do this; the user selects \texttt{fftOdd} within the \texttt{where} clause of \texttt{ffth} and selects \textit{Add Binding to runEval} from the HaRe drop down menu. This time HaRe augments the existing \texttt{runEval} monad by adding an additional binding, \texttt{fftOdd' <- rseq fftodd}, after the original binding of \texttt{fftEvn' <- rpar fftEvn}. Finally, the \texttt{return} statement is changed to return a tuple, where each component returns a binding within the monad. The pattern match is also changed to reflect this, as is the substitution of \texttt{fftOdd} for \texttt{fftOdd'} within the body of \texttt{ffth}. The modified code is now as follows:

\begin{verbatim}
ffth :: [ComplexF] -> [ComplexF] -> [ComplexF]
fifth xs us
| n>1 = (ffmups (replikate fftEvn') (ffmus us (replikate fftOdd')))
| n==1 = xs
where
\end{verbatim}
rvv = ffmus us (replikate fftOdd)
rvv1 = replikate fftEvn
fftEvn = ffth (evns xs) uEvns
fftOdd = ffth (odds xs) uEvns
uEvns = evns us
evns = everyNth 2
odds = everyNth 2 . tail
n = length xs

(fftEvn’, fftOdd’) = runEval $ do
  fftEvn’ <- rpar fftEvn
  fftOdd’ <- rpar fftOdd
  return (fftEvn’, fftOdd’)

Although this version of the code introduces parallelism at the expected points, it does not exhibit the desired behaviour: the parallelism generated from it is very small. The reason for this behaviour is that the default evaluation degree is used on the list arguments to rpar. This will only evaluate the top-level list cell in a parallel thread, generating little parallelism. By using HaRe when introducing parallelism, the programmer gets a warning about an undefined evaluation degree and is therefore automatically made aware of this problem, which is a common pitfall to inexperienced programmers. To achieve the desired behaviour, the user has to apply a “modify evaluation degree” refactoring on the two applications of rpar, for example selecting full evaluation (rdeepseq) on each of them. This results in the code below:

ffth :: [ComplexF] -> [ComplexF] -> [ComplexF]
ffth xs us
| n>1 = (ffmups (replikate fftEvn’) (ffmus us (replikate fftOdd’)))
| n==1 = xs
where
  rvv = ffmus us (replikate fftOdd)
  rvv1 = replikate fftEvn
  fftEvn = ffth (evns xs) uEvns
  fftOdd = ffth (odds xs) uEvns
  uEvns = evns us
  evns = everyNth 2
  odds = everyNth 2 . tail
  n = length xs

(fftEvn’, fftOdd’) = runEval $ do
  fftEvn’ <- (rpar ‘dot’ rdeepseq) fftEvn
  fftOdd’ <- (rpar ‘dot’ rdeepseq) fftOdd
  return (fftEvn’, fftOdd’)

215
4.2 Introduce Thresholding

After inspecting the timings of threads, the programmer has decided that it would be a sensible move to introduce a threshold for turning on the parallelism. This would allow granularity to be sufficiently high to make parallelism worth while. In order to do this, the programmer need only call the *Introduce Threshold* refactoring from HaRe, rather than manually introducing this threshold.

In order to perform the refactoring, first the programmer selects \( n \) in the `where` clause of `ffth`; here \( n \) will be used in our guard clause to compare against a threshold parameter. After choosing the *Introduce Threshold* refactoring from the HaRe drop down menu, HaRe prompts for a threshold value, where the programmer enters the value 2048. The result of the refactoring is as follows:

```haskell
ffth :: Integer -> [ComplexF] -> [ComplexF] -> [ComplexF]
ffth threshold xs us
  | n > threshold = (ffmups (replikate fftEvn')) (ffmus us (replikate fftOdd'))
    where
      (fftEvn', fftOdd') = runEval $ do
        fftEvn' <- (rpar 'dot' rdeepseq) fftEvn
        fftOdd' <- (rpar 'dot' rdeepseq) fftOdd
        return (fftEvn', fftOdd')
  | n>1 = (ffmups (replikate fftEvn')) (ffmus us (replikate fftOdd'))
    where
      (fftEvn', fftOdd') = runEval $ do
        fftEvn' <- rseq fftEvn
        fftOdd' <- rseq fftOdd
        return (fftEvn', fftOdd')
  | n==1 = xs
    where
      rvv = ffmus us (replikate fftOdd)
      rvv1 = replikate fftEvn
      fftEvn = ffth (evns xs) uEvns
      fftOdd = ffth (odds xs) uEvns
      uEvns = evns us
      evns = everyNth 2
      odds = everyNth 2 . tail
      n = length xs
```

The refactoring introduces the threshold as an argument to `ffth`, replacing all calls to `ffth` in the scope that it is defined with `ffth 2048`. The argument is added in the first position so that partial applications of the function can be taken into account. The `runEval` monad that was introduced in Section 4.1 is now moved to the `where` clause for the threshold guard, as the purpose of the threshold is to invoke parallelism. The remaining `n > 1` guard is then transformed to use an identity strategy instead, therefore running a sequential algorithm until the threshold is true.
5 Related Work

Despite the obvious advantages, there has so far been little work in the field of applying software refactoring technology to assist parallel programming. The earliest work on interactive tools for parallelisation stemmed from the Fortran community, targeting loop parallelisation [12]. These interactive tools were early transformation engines allowing users to manipulate loops in their Fortran programs by specifying what loops to interchange, align, replicate or expand. The interactive tools typically reported to the programmer various information such as dependence graphs, and was mainly applied to the field of numerical computation.

Recent work in the field includes Reentrancer [13]: a refactoring tool developed by IBM for making code reentrant. Reentrancer targets global data by making them thread-safe. Further recent work includes a refactoring approach to parallelism by Dig [14], targeted at introducing concurrency in Java programs by aiming to make them more thread safe, increasing throughput and scalability. Hitherto, Dig’s refactoring tool contains a minor selection of transformations including make class immutable, parallelise loop and convert HashMap to ConcurrentHashMap.

Software refactoring techniques have therefore only previously been applied in a very limited parallel setting: by applying simple transformations to introduce parallel loops and thread safety in OO programs. Currently, these approaches do not take any extra function properties into account, such as hardware characteristics, costing and profiling, for aiding the refactoring process. Furthermore, the techniques are rather limited to homogeneous architectures and OO languages, rather than applying general patterns to heterogeneous architectures.

6 Conclusions and Future Work

Parallelising a Haskell program is simple: all the programmer has to do is to introduce strategies, describing which expression could be evaluated in parallel. However, identifying the most useful sources of parallelism and tuning the performance of the initial parallel code can be tricky and often requires expertise in the parallel programming as well as an understanding of evaluation order and degree in Haskell.

At this point, the parallel refactorings for the GpH extension for Haskell help, because they encode expertise of these expects of the execution in an interactive tool, that guides the programmer through the process of adding parallelism and tuning performance. It warns the programmer to avoid common pitfalls, such as under-defining the degree of evaluation, and gives a structure way to performing common optimisations such as introducing thresholding to a parallel program.

We have demonstrated the usefulness of these refactorings on several small, and one realistic parallel application from the parallel section of the parallel nofib suite. Most of the refactorings presented in the paper have already been implemented in the HaRe framework for refactoring, and will be available as a HaRe snapshot once the final paper is ready.
This work is in fairly early stages of development and there are several directions for further work. Most of all we have to complete the implementations of all the refactorings described in this paper. There are several more refactorings that would be immediately useful for parallelism, or that would enable other refactorings by changing the code structure. We then plan to apply the re-factorings to some of the larger examples that we have studied in the past [1] and also to new application to assess the usefulness in the complete process of parallelisation. Another potential use case for such re-factorings would be the transformation of parallel programs from the original version of evaluation strategies [9] to the new version [1]. While this is mainly of interest for developers of new strategies, this can give insights in the advantages in the different formulations of strategies.

References

3. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1992)
Exploiting Parallelism by Customizing Evaluation

Silvia Clerici and Cristina Zoltan
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Barcelona, Spain

Abstract. NiMo is a totally graphic language from the family of Higher Order Typed languages with a strong Data flow inspiration. The interpreter is a specialized graph transformation system, and therefore the language operational semantics is given in terms of graph transformations. In NiMo parallelization is implicit and the evaluation policy is customizable following a process-centered approach. Here we explore some of the methodological possibilities that it opens. Some classical examples illustrate how combining modes greatly increases processor usage, decreases channel population, and achieves subnet synchronization in a very easy and intuitive way. We also present a stream programming technique and a real case application for generative and multistage-programming.

Keywords: Parallel Programming, Evaluation Policy, Data-Flow Languages, Functional Languages

1 Introduction

With the emergence of commodity multicore architectures, exploiting tightly-coupled parallelism has become increasingly important. Most of the parallelization efforts are addressed to applications that compute with large amounts of data in memory and in general have a regular behavior. In the scenario of tiny artifacts, interactive/reactive applications do not deal with huge amounts of data in memory, but streams of data. However they can still exploit several cores in a fine grain parallelization. Hence, finding simple ways to increase parallelism is now a matter of general and growing interest.

NiMo (Nets In Motion) is a graphic-functional-data flow language designed to visualize algorithms and their execution in an understandable way. The bi-dimensional representation displays the chain of process dependences revealing the implicit parallelism, and graphic execution helps us to understand where and when resources are used in the program, thus giving clues to optimize the solution. Programs are process networks that evolve showing their full state at each execution step. Processes are polymorphic, higher order and have multiple outputs. The language has a set of primitive processes well suited to stream programming and supports open programs and interactive debugging. The system provides an also graphic and incremental type inference system [1] that guarantees type-safeness by construction.
NiMo is a non-strict parallel language, but there are not explicit constructs to indicate parallelization. In the NiMo model all processes selected to act are supposed to execute in parallel at the same execution step. An execution step is a transition from one net to the next, where all the selected processes have produced the corresponding graph transformation. From the user point of view all of them have acted in parallel. The system provides facilities to measure the used resources (parallelism level, number of steps, number of processes, etc.).

The NiMo initial version followed a parallel lazy evaluation policy; nevertheless, the user could locally modify the evaluation order by setting an explicit requirement on any process. In the current version, a variety of modes determine the activeness level of each process. Modes can be globally or locally set for each process and dynamically changed giving the user a very intuitive and notably flexible way of customizing evaluation order. By tuning modes, the user can increase the number of processors that could act concurrently, regulate channel population or the evolution of the number of alive processes during execution. Furthermore, the evaluation modes could be used for deactivating subnets during experimentation or promoting speculative calculations, and to prevent evaluation of symbolic values. Since NiMo programs are graphs that evolve in execution, the operational semantics of the language was given in terms of graph transformation rules. It was presented in [2]. Here we discuss the possibilities opened by a flexible evaluation policy.

The next section gives a very brief introduction to the language constructs necessary to understand what follows. A more complete description of the language can be found in [3]. Section 3 summarizes the NiMo execution model, giving the repertory of process modes and a first example of their use. Section 4 presents a case study for the classical quicksort algorithm to illustrate how its performance can be drastically improved by only changing modes. Afterwards a further transformation increases even more the pipeline parallelism. Section 5 discusses the use of modes to handle symbolic execution and generative or multi-stage programming.

2 NiMo elements

NiMo programs are directed graphs with two kinds of nodes: processes and data items. Horizontal arrows represent channels of flowing data streams, and vertical arrows entering a process are non-channel parameters, which can also be processes. Processes can have any number of inputs and outputs, making the use of tuples unnecessary. There are neither patterns nor specific graphical syntax for conditionals. The main tokens are: rounded rectangles for processes, circles (or ovals) for constant values, black-dots for duplicators, and hexagons for data elements. Circles are labeled with their value for atomic types or with their names for symbolic constants of any type, even polymorphic. Hexagon labels are I, R, B, L and F for integers, reals, booleans, lists, and functional processes. Polymorphic data are labeled with “?”. The NiMo syntax makes intensive use of color. In hexagons and circles it indicates their type, in process names it denotes the
evaluation mode, and edges have a state shown as a colored diamond to indicate process activation or data evaluation degree. All the mentioned nodes are interfaces having typed (in/out) connection ports. Interfaces are dragged from a ToolBox and dropped into the workspace where the net is being built. Clicking on a pair of ports connects them with an edge if both types are compatible; otherwise a failure message is generated. When a process output is connected the diamond is white, incoming data items are connected with a green diamond. The user can change the white diamond of a process output to red in order to request the process to act, or it can be changed in execution by its precedent process. There are two kind of processes: the gray ones are built-in processes (bPs) for basic types and stream processing, and the white ones are user defined processes (nP s). The repertory of bPs includes multiple output versions of many Haskell prelude functions, as the process SplitCond, that splits its entry channel according to the condition stated by the process connected at its vertical entry\(^1\). Also, some basic processes have configurable arity, as a Map with n input and m output channels (generalizing map, zipWith and zipWith3 ), TakeWhile and Filter with n input and output channels, and an Apply process. There are also two kind of hexagons: list-item hexagons for channel elements and terminal hexagons corresponding to the net outputs. Since the flow is from right to left, they are the leftmost interfaces. Subnets connected to a terminal hexagon are considered productive even when incomplete because in NiMo a process can execute even though its non-needed inputs are left open. In execution all the non-productive subnets are deleted by the garbage collector.

2.1 An Example

Fig. 1 shows the fifth execution step of a prime numbers generator (Eratosthenes sieve algorithm). In this case, the result (leftmost hexagon, labelled L) is a

\[
\begin{align*}
\text{splitCond p l} &= (\text{filter p l}, \text{filter notp l}) \\
\text{notp x} &= \text{not(p x)}
\end{align*}
\]

\(^1\) In Haskell code splitCond p l = (filter p l, filter notp l) where notp x= not(p x)
list with two already calculated elements (light blue hexagons labeled I) with constant values 2 and 3. The remaining elements are to be produced by the (recursive) subnet *sieve* whose entry channel at this step has a first element with value 5 and the next ones are to be produced by the subnet *fromStep* generating the odd numbers. The **bPs** *SplitCond* and *Map* are marked with a red frame. In the next step these will execute in parallel. The subnets *sieve* and *fromStep*

![Diagram](image)

2: *sieve* (*fromStep* 3 2)

**Fig. 2. Net primes**

are net processes. Fig. 2 shows the initial net (and the equivalent Haskell code).

**nP**s are user-defined components whose interfaces (the white rounded-rectangles) are defined by means of a parameterization mechanism. The net in/out open ports that are to be considered as formal parameters or results are bound to the in/out ports of a configurable interface that is given a name. Later it can be imported to the Toolbox to be used as a process in a new net and so on, allowing incremental net complexity up to any arbitrary degree. At execution if the **nP** has to act, the white process interface is replaced by its associated net even if some of its parameters is not connected. Fig. 3 shows the net process definition

```
fromStep a b = ls where ls = a: map (+b) ls
```

![Diagram](image)

**Fig. 3. Net process definition for fromStep**

for *fromStep* and its correspondence with the equivalent Haskell code.
3 Executing nets. Process modes

In the initial version NiMo followed a parallel lazy policy. All processes acted only under demand (only when some of their outputs had a red diamond), except a distinguished one that continuously forced its provider to act. The user could also set a demand on a given process by changing to red some of its output diamonds. A required process without enough data demanded the necessary input providers to act and so on (red diamonds propagation). Now the evaluation policy is no longer uniform. Several modes of increasing activeness can be set globally (for all) or locally for each process and can also be changed during execution. Modes alter the process scheduling without changing the code (only the color of process names changes). The modes for basic process are:

- **Disabled**: the process is not able to execute (even if requested).
- **Demand-Driven**: The process is able to execute only if requested.
- **End-Driven**: The process is also able to execute as soon as it can end and disappear (for instance, a non-required map can execute whenever any of its input channels ends).
- **Data-Driven**: The process is able to execute as soon as it has enough data.
- **Weak-Eager**: the process is always able to execute or to request its needed input providers.

Net processes have three possible modes:

- **Disabled**: will never expand;
- **Demand-Driven**: only when requested;
- **Auto-Expand**: always applies its expansion rule.

When only the outermost processes (the ones nearest to the net outputs) are set to Weak-Eager, and all the other processes are set to Demand-Driven the net has a total lazy parallel behavior. In order to simplify the net, Demand-Driven bP processes can be set to End-Driven. Setting all processes as Data-Driven gives the usual semantics in data flow approach. But an eager semantics cannot be emulated in NiMo by only changing modes (additional red diamonds must be set), because NiMo is a non-strict language and Weak-Eager processes require only their needed inputs. The process-centered approach makes possible to have the advantages and overcome the drawbacks of all the standard evaluation policies. The following section shows how an appropriate combination of modes allows increasing the implicit parallelism, dealing with subnets synchronization and regulating channel population. The example illustrates the use of the system tools for measuring the execution behavior.

3.1 Experimenting with modes

Let’s consider again the prime numbers example in section 2.1. The net definition of sieve is shown in Fig. 4. A roughly equivalent Haskell code (since in NiMo there are not patterns nor special syntax for conditionals) is the following:
sieve [] = []
sieve (a:x) = a : sieve (filter (nodiv a) x)
nodiv a n = (mod n a) > 0

The *sieve*’s input list is the entry of its right-most process *Hd-Tl*. The head of the entry list is duplicated to be the first generated output of *sieve* and the (second order) parameter of *isDiv*. Process *SplitCond* splits the tail of the entry list into two channels, but the first (with those that satisfy the condition) is left open, then acting as a filter for the opposite condition. The resulting values are the entry for the recursive call of *sieve*, once proven the list is not empty by means of the conditional process *ifBool*. In fact one could argue that in this context it will never happen and we could eliminate the conditional process. That is true, but in NiMo a net process interface (not in Disabled mode) is replaced by its internal net definition whenever it is required to act, i.e. when its has a red diamond in any of its outputs, which persists in the bound internal process output. Hence, even not having yet a first element to act the *sieve* net would be expanded, then the same would occur with its internal *sieve* interface and so on, with the unnecessary waste of memory and screen. The conditional overcomes this problem\(^2\) because its condition only can be evaluated when a new value is produced by *SplitCond*. At this time the process *ifBool* disappears (together with its first and second parameters), *sieve* interface takes its place maintaining the red diamond in its output connection and it expands in the next step.

Hereafter we analyze how process modes impact the network performance. In the first experiment we choose a lazy policy, i.e. all processes (in the first level net and also in the subnets) are set to demand driven mode and an activator process is added to force a continuous demand in the *primes* net output. Fig. 5 shows the execution at the time when the prime 11 is generated. According to the system step counter this take place at step 89 and we can see two system resource viewers. The one on top shows that most of the time there is only

\(^2\) As happens with the Haskell patterns.
one process acting, except when fromStep also acts in parallel to produce the next odd number. The other one shows that the number of data and processes are almost the same. A lazy approach in general produces low parallelism but maintains the channels size bounded.

In the second try all (bPs) processes are set to Data-Driven. Fig. 6 also shows the execution at the time when the prime 11 is generated. In this case it is produced at step 39, and in the resource viewer on the top it can be seen that the parallelism level increased, being bounded by the number of already
generated primes. Also, we can see that the number of elements in the channel connecting subnets \textit{fromStep} and \textit{sieve} increases as well (let observe the graphic of process/data). At this time there are ten already generated odd numbers waiting to be treated. It happens because the productive rhythm of \textit{fromStep} is much higher than the consumption capacity of \textit{sieve}. In general, the Data-Driven approach, common in data-flow languages, increases the parallelism since processes act as soon as they get their data, but this behavior can saturate channels.

The way of keeping the advantages of both policies is to use a lazy approach only in the places where the consumer is not able to process at the same rate of its producer. In this case, changing the duplicator’s mode to Demand-Driven since it is the leftmost process of \textit{fromStep}. But this implies that the consumer process in \textit{sieve} has to be changed to Weak-Eager in order to activate the duplicator once having processed the previous element. The leftmost process in \textit{sieve} is \textit{Hd-Tl} but it dies after the first computation step, therefore the process that must keep the demand on \textit{fromStep} is \textit{SplitCond}. With just these changes we get a nearly equivalent amount of parallelism, since 11 is obtained in step 41 without overpopulating the channel.

4 Example: QuickSort NiMo Program

In this section we use a simple quicksort to show how modes for processes are determined. Due to experimentation, a new algorithm is presented along with the notion of active stream.

As remarked in [4], Quicksort remains one of the most studied algorithms in computer science. The version therein presented exploits concurrency without locking or explicit synchronization. The goal is obtained by using mutable store in a RAM memory. The algorithm is presented in Orc [5], which is a language based on a calculus with very few combinators; one of them is the parallel combinator. As we already said, in NiMo there is no explicit construct for signaling possible parallel process execution; processes can be executed in parallel if they are able to act\footnote{have all the required values and their modes allow them to act.} in the same computation step.

The classical simple QuickSort algorithm in Haskell code is

\begin{verbatim}
qsort [] = []
qsort (x:xs) = qsort less ++ [x] ++ qsort more
   where
      less = filter (<x) xs
      more = filter (>=x) xs
\end{verbatim}

Fig. 7 shows the equivalent NiMo net process definition where the conditional process \textit{ifbool} and the access process \textit{Hd-Tl} replace the pattern matching mechanism not present in NiMo. Reading the component from left to right, the concatenation process (++) brings together the elements that are smaller than
the pivot (if any) and the elements greater than or equal to the pivot. The input’s head is duplicated to become the pivot and also the second order parameter of SplitCond that splits the tail of the entry into the two segments. Each one is the entry for two recursive applications of QS, once proven they are not empty. If not, an empty list is produced as the result. If we set all the processes in Fig. 7 to Demand-Driven the net will have a lazy behavior. Hence, the program in Fig. 8 that uses QS has a process to force a continuous demand on the net. The entry list is a sample of 15 values in the best case, i.e values in a balanced tree shape. The execution of this version takes 303 steps.

If we set all bPs in the net QS to Data-Driven, leaving the internal QS processes as Demand-Driven, and change to red the diamonds in front of both ifBool, red diamonds will activate QS only if they need to be expanded. To run this modified version of QS, the activator process in Fig. 8 can be eliminated and the QS process must be set to Auto-Expand. By changing these modes, the number of steps for the same input is 61 instead of 303.

This version of QS exploits the implicit parallelism induced by the divide and conquer strategy. Looking at the algorithm execution, we can see that the process (++) treats its first operand element by element. All the values in the
first operand need to be produced before the process can glue the second operand to construct the result (this operation takes one execution step). In Fig. 9 we have the result of executing a single recursive call to $QS$ with the sequence 7,3,9,2,4,8,10 which corresponds to a balanced binary tree$^4$. The initial pivot is 7. For obtaining this picture we have executed the program setting as Disabled the $QS$ processes in the $QS$ definition, in order to block the activation of recursive calls.

We can eliminate the delay produced by the concatenation by transforming the net in what we call an active stream. Instead of constructing (using the concatenate process) the output stream, the net associated to the sorting procedure can be seen as a stream of processes and values i.e. an active stream. The definition for the new net process ($QS2$) is the one on Fig. 10.

Processes $QS2$ have two input parameters: the stream to be sorted (top right) and another input parameter which is the rest of the stream (bottom right). The process $QS2$ will be replaced by the ordered permutation of its first parameter and glued to its second parameter. The result is the sequence formed by ($QS2$) applied to the stream of values smaller than the pivot, followed by the pivot followed by another ($QS2$) having as its first parameter the stream of values greater or equal to the pivot and as its second the rest of the stream.

$^4$ The tree $[7,[3,[2],[4]],[9,[8],[10]]]$
In Fig. 11 we can see the sequence construction.

We compare the behavior of the three algorithms using sample inputs like a balanced binary tree, which is its best behavior, the sequence of 15 elements in strict descending order([15..1]) and the sequence already sorted ([1..15]).

Table 1. Comparing three quick sort

<table>
<thead>
<tr>
<th>Method</th>
<th>Input</th>
<th>Steps</th>
<th>steps for first</th>
</tr>
</thead>
<tbody>
<tr>
<td>QS-lazy</td>
<td>[1..15]</td>
<td>496</td>
<td>52</td>
</tr>
<tr>
<td>QS</td>
<td>[1..15]</td>
<td>119</td>
<td>49</td>
</tr>
<tr>
<td>QS2</td>
<td>[1..15]</td>
<td>119</td>
<td>48</td>
</tr>
<tr>
<td>QS-lazy</td>
<td>[15..1]</td>
<td>678</td>
<td>528</td>
</tr>
<tr>
<td>QS</td>
<td>[15..1]</td>
<td>146</td>
<td>133</td>
</tr>
<tr>
<td>QS2</td>
<td>[15..1]</td>
<td>121</td>
<td>121</td>
</tr>
<tr>
<td>QS-lazy</td>
<td>balanc tree</td>
<td>303</td>
<td>110</td>
</tr>
<tr>
<td>QS</td>
<td>balanc tree</td>
<td>61</td>
<td>55</td>
</tr>
<tr>
<td>QS2</td>
<td>balanc tree</td>
<td>60</td>
<td>51</td>
</tr>
</tbody>
</table>

Measures used in Table 1 are the total number of steps needed to sort the sequences, and the number of steps needed for the result to start flowing. In the case where the input sequence is ordered there is almost no difference between QS and QS2 because there is almost no delay motivated by concatenation. We see that when the input is the descending sequence (the worst case for quicksort method), the output for QS2 starts flowing much earlier and takes almost as long as sorting the entire sequence\(^5\).

5 Symbolic Computation and Generative Programming

The possibility of disabling processes has multiple applications. When the process is not yet defined it is the way to prevent it from being forced to act. In

\(^5\) This algorithm runs faster and even uses less space than the classical version also in Haskell.
addition, set to Disabled some processes allows partial testing of subnets, or executing the algorithm by levels enabling to reason about the net structure in the intermediate states. This section discusses other two additional uses of disabling processes for symbolic computation and generative programming.

Fig. 12. Symbolic execution

The first one is here illustrated with an example where equivalent codes are compared in an abstract way. In NiMo there are symbolic constants of any type, even polymorphic. Fig. 12 shows the initial results of evaluating $\text{map}(f \circ g)[x_1, x_2, x_3, x_4, x_5]$ and $\text{map} f(\text{map} g[x_1, x_2, x_3, x_4, x_5])$. In this case the Disabled processes $f$ and $g$ are being used as symbolic functional values; they do not have an associated net process definition.

Regarding generative programming the idea is to define a net that generates a second one with some Disabled processes. The resulting net is the desired program. In order to be executed it is only required to globally set the Disabled processes to another mode, and execution proceeds in the next step (multi stage execution) or else the resulting net can be stored to be run later. This technique has been used (in collaboration with the UPC team of the WISEBED project [6]) to generate different topologies for sensor networks of variable size and afterward simulating their behavior [7].

The example in Fig. 13 corresponds to the classical problem of calculating average temperature using meteorological sensors. The initial net (on the left) is parameterized by the topology (ring, tree, star) and number of sensors. Its execution generates a network with the desired topology and all sensors having Disabled mode so that no one executes until the network is completely constructed since all of them have to produce the average temperature at the same time. Each sensor process is parameterized by the delay to produce its first output according to its position. It is simulated by means of a chain of activator processes (taking one execution step each). Generation also includes processes that simulate the input from the environment for each sensor. The net on the
right of Fig. 13 shows the fifth step of the generation stage. The initial net evolves until no more processes can act. Now by only changing globally the modes in Fig. 14 the execution becomes the sensor network simulation. Fig. 15 shows the net state when two values were produced in each sensor output. Fig. 16 shows the generation stage result when the topology is a binary tree with depth 3.

Another example of multistage execution, in this case combining symbolic and numeric computation, is the generation of the Newton’s binomial expansion for integer constants with symbolic values $a$ and $b$ and numeric value for the exponent. The arithmetic operator processes (+, ∗ and ^) with symbolic parameters are set to Disabled mode. The execution stops once generated the binomial
expansion. If we run the same net with numeric values instead of $a$ and $b^6$, once generated the expansion we can change to Data-Driven the Disabled mode for all the processes in the resulting net, the execution continues and the numeric final value is produced.

---

6 Or if we replace all occurrences of values $a$ and $b$ in the expansion.
In programming languages exploiting parallelism, there are constructs or annotations that the programmer uses to parallelize or help the compiler to do so. The presentation of the program changes with these additions. With this concern in mind, [8] introduces strategies. In this approach “A function definition is split in two parts, the algorithm and the strategy, with values defined in the former being manipulated in the latter”. Strategies allow establishing individual evaluation degrees for elements present in function definitions. They are associated to data types and for instance can describe how deeply individual elements of a list are to be evaluated. Strategies are used also for controlling parallelism.

In NiMo parallelism is given by the net structure. Different branches can be executed in parallel, different parameters of a process can be evaluated in parallel. Pipeline parallelism occurs when in a chain of processes several ones acquire enough data to act. The NiMo programmer strategy focuses on determining the activeness degree for each process. Also, as the annotations are just the color on the names of processes or on diamonds, the structure of the program remains exactly the same keeping the mental model. The language minimizes the need for programmer involvement in identifying parallelization. Using the classification given in [9] we can say that in NiMo the programmer could design its algorithms being aware that the execution will be done in a parallel model of execution (second level of abstractness), but the algorithm also runs and is efficient in a single processor model of computation. The first goal is to obtain a correct solution. Users have total control over their code and the state of its execution, they can interactively modify any program element undo steps, and store any state as a new net, therefore the system acts as an online tracer and debugger. On this regard, Disabled mode allows testing pieces of nets or visualizing the algorithm structure in intermediate states. Afterwards, the program can be tuned by observing the net behavior and the resource system viewers in order to accelerate execution and/or reduce memory usage.

On the other hand, in NiMo processes with unbound or symbolic parameters can execute. Disabling processes is also the way of integrating symbolic computation in the same framework and handling incompleteness. It also provides the means for generative programming and multi-stage execution.

The term “generative” in software development is associated to a system-family approach, which focuses on automating the creation of system-family members[10]. A variant is multi staged programming [11] which makes simplifications of generic programs developed using good abstraction mechanisms. This improves the efficiency of the resulting programs. Both approaches need language extensions (meta-programming or Domain Specific Languages).

In our context the term refers to having a generating net whose final result is a new executable net. If the original net is parameterized then there is a family of resulting nets. In general, the existence of Disabled processes may stop the
execution, which can be resumed by only changing their modes.

As a final conclusion we can say that by setting modes the user can modify the parallel scheduling in a simple and intuitive way since:

– modes define several levels of increasing activity (vs. the usual dichotomy lazy/eager).
– the process-centered approach allows detailed customizable evaluation
– modes can be dynamically changed global or locally

These characteristics give a notable flexibility in program tuning for resource usage optimization.

References

Abstract. Although thread-safe priority queues are fundamental building blocks for many parallel algorithms, there are currently no implementations available in Haskell. An efficient structure to implement Priority queues is the skiplist, which is a multi-level linked-list with shortcuts. We developed three thread-safe skiplist variants, based on locks, software transactional memory and atomic compare-and-swap, respectively. In our benchmarks, the lock-based and compare-and-swap variants scaled about equally well, while the transactional variant was by an order of magnitude slower or on par, depending on workload.

1 Introduction

Priority queues are well-known data structures to store and retrieve elements from some ordered set [1]. They support at least two operations: insert adds an element to the queue, and deleteMin returns the minimal element. Priority queues are used in many parallel algorithms, where they need to be thread-safe.

Unfortunately there are currently no thread-safe priority queue implementations available in Haskell. The main contribution of this paper is providing implementations based on skiplists [2], which are multi-leveled linked-list data structures with shortcuts to randomly chosen elements. Skiplists are superior in performance to classic approaches such as the thread-safe binary heap implementation of Hunt [3], as has been shown in [4–6] for both locked-based and lock-free versions. We implemented three variants of skiplists. One ensures thread-safety with explicit locking, while the others are lock-free and based on software transactional memory and atomic compare-and-swap operations, respectively. We benchmarked the different versions using random (but reproducible) insert and deleteMin operations with and without computational load in-between. Unlike many other implementations of thread-safe priority queues which only skimmed over duplicate key handling, our queues fully support duplicates, i.e., multiple elements may have the same key, as they are frequently needed in practice. We tested the correctness of our implementations with Johnson’s algorithm [7] for the single source shortest path problem for sparse graphs. To briefly state
our results, the lock-based and atomic compare-and-swap variants scale comparable under computational load, while the performance of the transactional variant heavily depends on the amount of additional computational workload, and ranges from that of the other variants to some orders of magnitude slower.

The full source code of our implementations, as well as the shell scripts for benchmarking and the raw benchmark results can be found at [8].

The rest of the paper is structured as follows. Section 2 gives an introduction to a sequential Skiplist implementation. Section 3 describes the different concurrent skiplist variants in detail. Section 4 explains our benchmarks and discusses the experimental results. Section 5 reviews related work and Section 6 concludes and gives an outlook to future work.

2 Skiplists

In this section we briefly explain the non-optimized version of the sequential skiplist data structure, first described and analyzed by Pugh [2]. For simplification, we postpone details of handling duplicates to the next section which introduces the concurrent versions.

A skiplist is basically an ordered linked-list with additional randomly chosen inserted shortcuts (see Figure 1). The shortcuts speed up searches and thus also inserts- and deletes by allowing to skip over a number of elements in one step (see Figure 2). The search operation starts at the initial node of the list at its highest level. It walks through nodes on the particular level until the following node has a larger or equal value. In this case search continues on the preceding level. The insert and delete operations use the remembered predecessors of each level to perform the well-known operations for linked-lists. The deleteMin operation used in a priority queue can be implemented by deleting and returning the first reachable element on level 1. On insertion, the height of each node is randomized such that 50% have height 1, 25% have height 2 and so on. That way, skiplists have a probabilistic time complexity of $O(\log n)$ for search, insert and delete and a worst-case time complexity of $O(n)$, where $n$ is the number of stored elements.
Fig. 2. Search for the element with key 5. By using the shortcut on level 3, keys 1 and 2 are skipped. The dashed arrows show the search order, the gray-shaded boxes the predecessors of the found node on each level. They are needed for the various linked-list operations.

3 Concurrent skiplist variants

In this section we first explain the PriorityQueue typeclass common to all implementations, and then describe our three variants of concurrent skiplists. The following presentation is occasionally simplified; further details can be found in the source code or the referenced papers, respectively.

3.1 The PriorityQueue typeclass

To ease experimentation and later usage, we defined a common typeclass with functional dependencies which is implemented by all skiplist variants:

```haskell
class Ord p => PriorityQueue pq p v | pq -> p, pq -> v where
deleteMin :: pq -> IO (Maybe (p,v))
insert :: p -> v -> pq -> IO ()
```

The deleteMin function returns the key and value of the minimal element or Nothing if the queue is currently empty. The insert function adds a key and its value to the queue. In contrast to many other concurrent versions of skiplist-based priority queues we decided support duplicates, although it increases the difficulty of a correct implementation. The motivation was that many real-world problems, e.g. Johnson’s algorithm, need duplicates to function correctly. All implementations can be modified to simply overwrite the value when a given key already exists.

3.2 Lock-based Skiplists

Our lock-based implementation is based on the original lock-based concurrent version of Pugh [9]. This version has the advantage that a thread only needs to lock a small portion of the skiplist and only for a short time to insert or delete a node, and that traversal does not need locking at all. Base types are shown in Figure 3. Each skiplist is initially empty and all pointers of the beginning node skipBegin point to the end node skipEnd. Each Node holds its key-value pair (if it is not the beginning or end node), a unique id to handle duplicates, a
data (Ord p) => Skiplist p v = Skiplist {
    skipBegin :: Node p v,
    skipEnd :: Node p v
}
data (Ord p) => Node p v = Node {
    nodeValue :: NodeValue p v,
    nodeId :: Unique,
    nodeGarbage :: IORef Bool,
    nodePointer :: Pointers p v,
    nodeLevel :: Lock Level,
}

data NodeValue p v = Value (p,v) | Begin | End deriving Eq

type Level = Int

type Pointers p v = IOArray Level (Lock (Node p v))

data Lock a = Lock {
    lockValue :: IORef (Maybe a),
    lockLock :: MVar ()
}

Fig. 3. Base types for the lock-based skiplist implementation.

deletion marker to signal deletion to other threads, its height, and references to its successors at each level.

Pugh’s algorithm requires both the support of concurrent read access to shared values and their exclusive locking such that other threads trying to lock are blocked (but are still allowed to read the value). We defined a type Lock which encapsulates a (possibly empty) modifiable value in an IORef, and use a MVar to achieve the blocked locking, as e.g. described in [10]. Since the (sequential) insert and delete operations work on a level by level basis, locking is mainly used to control the shared access to the successor pointers of each node as follows. Both operations first create a vector of predecessor nodes (as in the sequential case). The insert operation starts to insert the created node level by level into the skiplist, beginning with level 1: after locking the successor pointer of the predecessor on the current level, it sets the pointer to the newly created node and the successor of the new node to the former node (see Figure 4). The delete operation starts by checking the current status of a node’s possible deletion: if nodeGarbage is false, no other thread deletes the node and it is set, otherwise deletion is stopped. The found node is then removed level by level, but starting with the highest one. This is motivated by the fact that pointers on the additional levels larger than 1 are solely shortcuts and removing them does not lead to a (partially) corrupted skiplist. A node is thus deleted when it is removed from level 1. Since other threads could possibly be traversing through the node currently being deleted, deleted pointers do backreference their
successor pointers to their predecessor to allow correct continuous traversal (see Figure 5). The operation to receive the minimal element (called deleteMin in

the typeclass) simply reads the key of the successor of the beginning node and tries to delete it. If it is successful, this element can be returned. If not, another thread deleted the element simultaneously and the operation is restarted.

Since the insertion is done level-wise and threads are allowed to insert key-value pairs with the same key (duplicates), there is no guarantee of the order of elements on each level (see Figure 6). When a particular key is deleted it is

![Fig. 6. a) External representation of a skiplist with two keys 2 b) Internal representation. Values in brackets show the unique identifiers.](image)

crucial that the correct predecessor pointers on each level are modified. To solve
this problem we add an unique identifier to each node, such that a search can find all correct predecessors on each level (see Figure 7).

![Diagram showing search for correct predecessors](image)

**Fig. 7.** By using unique identifiers, the predecessors for key 2 with identifier (1) can be found. They are marked in gray.

### 3.3 Software Transactional Memory based Skiplists

Since lock-based programming has well-known drawbacks like deadlocks and difficult debugging, interest in alternative approaches has risen in the last years. Software Transactional Memory (STM), coming historically from database research, uses the idea of *transactions* to allow concurrent access to shared data structures without explicit locking and unlocking [11]; instead, functions access shared data structures in transactions and the runtime systems takes care of consistency issues, e.g. by restarting transactions in case of conflicts. To restrict the number of variables that the parallel runtime systems need to take care of, (transactional) shared values in Haskell are wrapped in a `TVar` and their use (i.e. both reading and writing) is only allowed inside a STM monad [12]. We developed two variants using STM, a naive one where the whole functionality for each operation is enclosed in one atomic block and a more sophisticated one where we tried to divide the different actions of each operation into independent blocks.

The data structure for this implementation is similar to the lock-based one shown in Figure 3. Instead of using `IORefs` and `MVars` to control mutable variables and locks, `TVars` are used exclusively. In the naive approach there is no need for a shared variable containing the deletion state since this is handled implicitly by the underlying transaction model.

The naive approach allows to easily model the sequential version, since the algorithms and approaches can be transcribed directly by simply changing mutable variables to `TVars` and enclosing the operations using `atomically`. As we discuss and explain in Section 4, the performance of this approach is comparably bad due to the possible long duration of each transaction and therefore high probability of conflicts and restarts.

A more reasonable approach is to divide the transactions into independent sub-transactions and thus to partially remodel the lock-based variants. Both the insert and delete operations have now two phases, transactionally independent from a STM point of view: in the first phase the predecessors for the correct position are found and, in the case of deletion, the corresponding node is marked as to be deleted. In the second, the new node is inserted at the correct position...
or removed, respectively; for each level a new transaction is started. Although it would be possible to further dissect the level-based transactions, i.e. approaching the lock-based solution further, this would (in our opinion) conflict with the overall idea of transaction-based synchronization and its promise to simplify parallel programming.

3.4 Skiplists based on atomic compare and swap operations

While lock-based implementations of thread-safe data structures are conceptually easy to comprehend, they imply the problems mentioned in the beginning of the last section. In contrast to transactional-based synchronization, lock-free synchronization with atomic compare-and-swap operations promises to deliver better performance due to the greater control of the operations. We used ideas for a lock-free implementation of linked lists from [4, 13]. To guarantee consistency of shared variables, atomic compare-and-swap (CAS) and test-and-set (TAS) operations are used. A CAS checks if the shared variable contains a particular value (usually the value of the variable read previously) and if and only if it does, swaps it with a new one. A TAS applies a function to the current value and swaps with the new value if the return value of the function is true. In both cases a returned flag indicates success. Haskell code for both CAS (from [14]) and TAS for modifying shared \texttt{IORef}s is shown in Figure 8.

\begin{verbatim}
atomCAS :: Eq a => IORef a -> a -> a -> IO Bool
atomCAS ptr old new = do
    atomicModifyIORef ptr $ \cur ->
    if cur == old then (new, True) else (cur, False)

atomTAS :: Eq a => IORef a -> (a -> Bool) -> a -> IO Bool
atomTAS ptr test new = do
    atomicModifyIORef ptr $ \cur ->
    if test cur then (new, True) else (cur, False)

Fig. 8. Atomic compare-and-swap and test-and-set functions.
\end{verbatim}

Types are slightly modified as compared to Figure 3. Since we are not able to lock nodes currently being changed but still need a mechanism to at least check their modification, we extend a reference (pointer) with a flag; many low-level implementations, e.g. [4, 15, 13], instead modify the least significant bits of a memory address. This architecture- and compiler-dependent approach is obviously not directly possible in Haskell with \texttt{IORef}s. Instead, a pointer now contains a tuple as shown in Figure 9 and is changed as described below.

The insert and delete operations use methods described in [13] for handling lock-free linked lists. We extend them to work on multiple levels (i.e. deletion works level-wise downwards and insertion upwards) as explained in Section 3.2.
data Mark = Marked | Unmarked deriving Eq

\[
\text{type \sample{Pointer} p v = (Mark, Node p v)} \\
\text{type \sample{Pointers} p v = IOArray Level (IORef (Pointer p v))}
\]

Fig. 9. Modification of pointers to allow annotation with a 'currently modified'-flag.

Both operations start by creating the vector of predecessors and then working level-wise in the specified direction. While insertion is similar to Figure 4, deletion using CAS is more complicated, since interfering insertions can lead to corrupt data structures (see Figure 10). The solution is to split the deletion in two phases: one marks the node to be deleted logically, such that concurrent insertions skip these nodes, and the following phase removes the node physically. The delete operation works therefore as follows: first, the node to be deleted is searched and logically deleted by marking the successor pointer of its predecessor using CAS; it repeats if the CAS fails. It is then physically deleted (also using CAS) by setting the pointer of the predecessor node to the successor of the deleted node, if possible. If a concurrent operation changed the pointers and the CAS failed, delete removes the node by searching for the key as indicated below.

The key function used by both operations is the search for the predecessor and successor of a node: while searching for these nodes the search also deletes...
nodes that it traverses and which are marked for removal by other threads (see Figure 11).

![Diagram](image_url)

**Fig. 11.** Representation of deleting traversed nodes in a search for a node with key 3. a) initial state. The marked nodes were marked by other threads b) state after traversal. The search function returns the beginning node as the predecessor and node 4 as the successor. Non-traversed nodes (e.g. node 5) are not influenced.

### 3.5 Checking implementation correctness

Developing concurrent algorithms is tremendously difficult. Hard to find errors, e.g. those corrupting the data structure instead of leading to a clearly visible error (like a deadlock) can be easily overlooked. While proving the correctness of an actual implementation is desirable, it is still a topic of ongoing research.

To challenge the different skiplist versions and at least test their desired functionality, we implemented Johnson’s algorithm [7], which calculates the single source shortest path (SSSP) in sparse graphs in parallel using a thread-safe priority queue as its basic data structure. For reference, we developed a simple sequential implementation and compared the outcomes. We ran each test a few hundred times with different seeds and graph sizes for all three skiplist implementations. We did not use the algorithm for performance measurement since efficient parallel formulations of Johnson’s algorithm are a difficult problem on their own [16, 17].

### 4 Benchmarks

We ran experiments on a 2.2 GHz 8-core AMD Opteron 875 with 16 GB RAM running RedHat Enterprise version 4 with Linux-kernel 2.6.18. Due to restrictions of the installed linux distribution we were only able to run GHC 6.12.1. Interpretation of the experimental outcomes was limited, since there is no parallel profiler for Haskell’s parallel runtime system, and Threadscope [18] primarily provides information regarding the garbage collector and overall processor utilization.

In our benchmarks each concurrent thread performs 300 initial insertions, followed by 3000 operations, chosen with a 50% probability to be deleteMin or insert. Keys are randomly chosen from the interval \((0, 10^6N)\) with \(N\) being the
number of threads. These values and probabilities were taken from other papers such as [4]. We re-run each benchmark 10 times and report the average execution times. Since the thread-specific random number generators are (re-)initialized with the same initial (but thread-specific) seed, the same sequential operations are performed in all runs and implementations, guaranteeing reproducibility. We performed three experiments with varying workload after each delete operation:

In the most realistic scenario, a reasonable amount of computational work is done after each deleteMin by having the thread that received value $n$ computes $\pi$ on $\frac{n}{20000}$ digits (for $n = 10^6$ calculation takes 0.002s). Results are shown in Figure 12; keep in mind that the time-axes in all figures are logarithmic. In this scenario, both the lock-based and compare-and-swap variants are an order of magnitude faster than the transactional variants and scale equally. The improved implementation (STM 2) performs noticeably better than the naive one (STM 1).

To challenge the implementations and see the influence of high contention, we omitted the workload in a second scenario. Results are shown in Figure 13. Not surprisingly, the transactional versions perform quite bad since transactions are restarted either on every conflict or for conflicts on each particular level. The compare-and-swap variant performs up to a factor of 21 better than STM 1 and 6 than STM 2 (depending on the number of cores), since modifications of the skiplist do not force a restart of the whole operation. The lock-based implementation performs best. We think that the reason for the difference between the
lock-based and all lock-free implementations lies in the fact that a locked node is a clear signal to other threads to either wait (on insertion) or continue search (on deletion), while in the lock-free variants many threads are interfering and need to restart the transaction or repeat the compare-and-swap, respectively.

Since contention plays a major role in the performance of the lock-free variants, it is interesting to look at the scalability when its influence is reduced: By increasing the amount of computational work to $\frac{n}{2500}$ (about 0.03s for $n = 10^6$), the performance of all variants becomes nearly identical (see Figure 14).

Summarizing, in scenarios with realistic workloads, the lock-based and compare-and-swap variants perform equally well while the transactional-based ones are an order of magnitude slower. In the pathological case of continuously accessing the priority queue, the lock-based variant performs appreciably better than both lock-free variants. When the influence of contention is reduced, all variants scale about equally.

5 Related work

Since priority queues are the building blocks for many parallel algorithms, an extensive amount of research has been done for their performant and thread-safe implementation. Traditionally, they are implemented using binary heaps [1]. A well-known parallel version which uses locks is described by Hunt in [3]. A general survey of different possibilities for thread-safe priority-queue implementations

![Fig. 13. Scalability of priority queue implementations without computational load.](image-url)
Fig. 14. Scalability of priority queue implementations with larger computational load.

(coarse locks, Hunt heaps, parallel fibonacci heaps, lock-free skiplists, quantitizing queues and transactional versions) and a discussion of their properties can be found in [19].

Skiplists, first described by Pugh are a performant alternative to binary heaps. Their first description and sequential pseudocode can be found in [2]. The same author later developed a lock-based concurrent version [9]. Both implementations showed a superior performance compared to binary heaps. An extensive discussion of the performance characteristics for lock-based priority queues using skiplists can be found in [5]. An algorithmic performance analysis for lock-free linked lists (and an outlook on skiplists) is described in [4], experimental results and a comparison on the performance of lock-free implementations with lock-based variants (Hunt, Lotan-Shavit) in [6]. Both works build on the implementation of lock-free linked lists as described in [13], which itself was an algorithmic improvement for the original work on this subject [15]; Sulzmann et al. analyzed the performance of concurrent linked list implementations in Haskell [14]. Like our work, it discusses synchronization mechanisms using MVars, IORefs and STM. In contrast to the results discussed and explained in Section 4, the lock-free implementations outperforms the other variants.

Although priority queues are fundamental for many parallel algorithms, there is as far as we know no prior literature about thread-safe priority queue implementations in Haskell. One possible reason is that they still work on a fairly low abstraction level. Haskell provides many alternatives which allow a more abstract formulation of parallel algorithms for the price of not being able to
transcribe well-known solutions found for traditional parallel systems. Intel’s Concurrent Collections provide a graph-based framework for formulating parallel algorithms[20], data parallel Haskell allows to implicitly evaluate parallel array algorithms[21] and the par-functions allows semi-implicit parallelization of functions, such that the compiler takes care of the (manual) synchronization done in this work[22].

6 Conclusion and Future Work

We have implemented thread-safe skiplists in Haskell, based on lock-based and lock-free synchronization techniques. Our lock-based variant used MVars, whereas the lock-free variants used software transactional memory (STM) and atomic compare-and-swap operations, respectively. To further examine the STM variant, we developed two versions: the naive version was a straightforward transformation of the sequential implementation, the other one dissected the transactions into independent parts. We checked the correctness of all variants by running Johnson’s algorithm for the single source shortest path problem on sparse graphs and comparing the results with a reference implementation. Each implementation was benchmarked using synthetic randomized operations that allowed reproducible results. The lock-based variant was significantly faster than its lock-free atomic compare-and-swap counterpart, although the advantage shrunk when the workload of each extracted element became large enough. The scalability of both transactional-based variants depended even more on the workload: if the load was small and contention therefore high, these variants were some order of magnitude slower, although the dissected one was faster. If the contention was low, their performance was comparable.

Future work may address different related topics: first, despite their sole use as priority queues in this work, skiplists are general dictionary structures. A comparison of their concurrent performance for arbitrary insertions and deletions seems interesting. Second, implementing parallel real-world applications which use priority queues can lead to further optimization possibilities for the different variants. Third, implementing and analyzing other thread-safe data structures (e.g. with finger trees as their underlying model) looks interesting. Fourth, the current rise of (experimental) manycore architectures poses the question of the scalability of our implementations to many more cores. This could be examined using Intel’s ManyCore Lab[23] and its experimental Haskell support.

References

Adams’ Trees Revisited
Correct and Efficient Implementation

Milan Straka
fox@ucw.cz
Department of Applied Mathematics
Charles University in Prague, Czech Republic

Abstract. We present a correct proof of Adams’ trees of bounded balance, which are used in Haskell to implement Data.Map and Data.Set. Our analysis includes the previously ignored join operation, and also guarantees trees with smaller depth than the original one. Because the Adams’ trees can be parametrized, we use benchmarking to find the best choice of parameters. Finally, a saving memory technique based on introducing additional data constructor is evaluated.

1 Introduction

Adams’ trees, or trees of bounded balance $\omega$, shortly BB-$\omega$ trees, are binary search trees introduced in [1] and [2]. In order to guarantee an asymptotically logarithmic depth of these trees, the size of a subtree is stored in every node. This information is useful not only for rebalancing, but also for other operations – the size operation runs in constant time, we can locate the $i$-th smallest element of the tree in logarithmic time, and so on.

BB-$\omega$ trees are used in Haskell to implement the Data.Map and Data.Set modules, which are part of the standard data structure library containers [10]. BB-$\omega$ trees are also used in data structure libraries in Scheme containers and SML. According to the measurements in [9], their performance is comparable to other alternatives such as AVL trees [3] or red-black trees [4].

Although the implementation of a BB-$\omega$ tree is quite simple, proving its correctness is tricky. The original proof in [1] has several serious flaws – it wrongly handles delete, it does not consider join and does not use the fact that the sizes of subtrees are integers.

Our contributions are as follows:

- We present a correct proof of BB-$\omega$ trees, including the previously ignored join operation. This new analysis guarantees trees with lower depths than the original one.
- We investigate the depth of BB-$\omega$ trees.
- Because the BB-$\omega$ trees are parametrized, we perform several benchmarks to find the best choice of parameters.
- In order to save memory, we evaluate the technique of introducing additional data constructor representing a tree of size one. This allows us to save 20-30% of memory and even decreases the time complexity.
2 BB-ω trees

We expect the reader is familiar with binary search trees, see [6] for a comprehensive introduction.

Definition 1. A binary search tree is a tree of bounded balance ω, denoted as BB-ω tree, if in each node the following balance condition holds:

\[
\begin{align*}
\text{size of the left subtree} & \leq \omega \cdot \text{size of the right subtree}, \\
\text{size of the right subtree} & \leq \omega \cdot \text{size of the left subtree}, \\
\text{if one subtree is empty, the size of the other one can be 1.}
\end{align*}
\]  

If the balance condition holds, the size of a tree decreases by at least the factor of \(\frac{\omega}{\omega + 1}\) at each level, so the maximum depth of a BB-ω tree with \(n\) nodes is bounded by

\[\log_{(\omega+1)/\omega}(n) = \frac{\log_2 n}{\log_2(1 + 1/\omega)}.
\]

Detailed analysis is carried out in Section 6.

The exception for empty subtrees is not elegant, but from the implementation point of view is of no concern – empty subtrees are usually represented by a special data constructor and are treated differently anyway. Nevertheless, some modifications to the balance condition have been proposed to get rid of the special case – most notably to use the size of a subtree increased by one, which was proposed in [8]. We therefore define generalized version of balance condition, which comprises both these cases:

\[
\begin{align*}
\text{size of the left subtree} & \leq \max(1, \omega \cdot \text{size of the right subtree} + \delta), \\
\text{size of the right subtree} & \leq \max(1, \omega \cdot \text{size of the left subtree} + \delta).
\end{align*}
\]  

The parameter \(\delta\) is a nonnegative integer and if it is positive, the special case for empty subtrees is no longer necessary. Notice that the definition with sizes increased by one is equivalent to the generalized balance condition with \(\delta = \omega - 1\).

An implementation of a BB-ω tree needs to store the size of a subtree of every node, which results in the following data-type:

```hs
data BBTree a = Nil -- empty tree
  | Node -- tree node
    (BBTree a) -- left subtree
    Int -- size of this tree
    a -- element stored in the node
    (BBTree a) -- right subtree
```

We also provide a function \textbf{size} and a smart constructor \textbf{node}, which constructs a tree using a left subtree, a key, and a right subtree. The balance condition is not checked, so it is upon the caller to ensure its validity.

```hs
size :: BBTree a -> Int
size Nil = 0
size (Node _ s _ _) = s

node :: BBTree a -> a -> BBTree a -> BBTree a
node left key right = Node left (size left + 1 + size right) key right
```
3 BB-ω tree operations

Locating an element in a BB-ω tree works as in any binary search tree:

```haskell
lookup :: Ord a => a -> BBTree a -> Maybe a
lookup k Nil = Nothing
lookup k (Node left _ key right) = case k ‘compare’ key of
  LT -> lookup k left
  EQ -> Just key
  GT -> lookup k right
```

When adding and removing elements to the tree, we need to ensure the validity of the balance condition. We therefore introduce another smart constructor `balance` with the same functionality as `node`, which in addition maintains the balance condition. To achieve effectiveness, certain conditions apply – the `balance` function can be used only on subtrees that previously fulfilled the balance condition and since then one element was inserted or deleted.

With such constructor, the implementation of `insert` and `delete` is straightforward. Assuming the `balance` constructor works in constant time, `insert` and `delete` run in logarithmic time.

```haskell
insert :: Ord a => a -> BBTree a -> BBTree a
insert k Nil = node Nil k Nil
insert k (Node left _ key right) = case k ‘compare’ key of
  LT -> balance (insert k left) key right
  EQ -> node left k right
  GT -> balance left key (insert k right)

delete :: Ord a => a -> BBTree a -> BBTree a
delete _ Nil = Nil
delete k (Node left _ key right) = case k ‘compare’ key of
  LT -> balance (delete k left) key right
  EQ -> glue left right
  GT -> balance left key (delete k right)

where glue Nil right = right
  glue left Nil = left
  glue left right
  | size left > size right = let (key’, left’) = extractMax left
  in node left’ key’ right
  | otherwise = let (key’, right’) = extractMin right
  in node left key’ right’

extractMin (Node Nil _ key right) = (key, right)
extractMin (Node left _ key right) = case extractMin left of
  (min, left’) -> (min, balance left’ key right)

extractMax (Node left _ key Nil) = (key, left)
extractMax (Node left _ key right) = case extractMax right of
  (max, right’) -> (max, balance left key right’)
```

When representing a set with a binary search tree, additional operations besides inserting and deleting individual elements are needed. Such an operation
is \textbf{join}. The \textbf{join} operation is also a smart constructor – it constructs a tree using a key and left and right subtrees. However, it poses no assumptions on the sizes of given balanced subtrees and produces a balanced BB-\(\omega\) tree.

In order to implement this operation we once again utilize the \textbf{balance} constructor, therefore changing its assumptions – the \textbf{balance} can be used on subtrees, that previously fulfilled the balance condition and since then one \textbf{insert}, \textbf{delete} or \textbf{join} operation was performed.

Having improved the balance operation, it is trivial to implement \textbf{join}. Once again, assuming balance works in constant time, \textbf{join} runs in logarithmic time.

\begin{verbatim}
join :: BBTree a -> a -> BBTree a -> BBTree a
join Nil key right = insertMin key right
    where insertMin key Nil = Node Nil 1 key Nil
        insertMin key (Node l _ k r) = balance (insertMin key l) k r

join left key Nil = insertMax key left
    where insertMax key Nil = Node Nil 1 key Nil
        insertMax key (Node l _ k r) = balance l k (insertMax key r)

join left@(Node ll ls lk lr) key right@(Node rl rs rk rr)
    | ls > omega * rs + delta = balance ll lk (join lr key right)
    | rs > omega * ls + delta = balance (join left key rl) rk rr
    | otherwise = node left key right
\end{verbatim}

\section{Rebalancing BB-\(\omega\) trees}

In order to restore balance, we use standard single and double rotations. These are depicted in Fig. 1. The code for these rotations is straightforward, the \textbf{L} or \textbf{R} suffix indicates the direction of the rotation (both rotations in the Fig. 1 are to the left).

The \textbf{balance} function restores balance using either a single or a double rotation – but a question is which one to choose. If we perform a left rotation as in Fig. 1, a single rotation leaves the left son of the right subtree unaffected, but a double rotation splits it into two subtrees. Therefore we choose the type of a rotation according to the size of the left son of the right subtree.

Formally, we use a parameter \(\alpha\)\(^1\), which we use as follows: When we want to perform a left rotation, we examine the right subtree. If its left son is strictly smaller than \(\alpha\)-times the size of its right son, we perform a single rotation, and otherwise a double rotation. The implementation follows:

\begin{verbatim}
balance left key right
    | size left + size right <= 1 = node left key right
    | size right > omega * size left + delta = case right of
        (Node rl _ _ rr) | size rl<alpha*size rr -> singleL left key right
        | otherwise = doubleL left key right
\end{verbatim}

\(^1\) Our \(\alpha\) differs from [1], in the sense that our \(\alpha\) is the inverse of \(\alpha\) from [1].
5 Choosing the parameters $\omega$, $\alpha$ and $\delta$

We call the parameters $(\omega, \alpha, \delta)$ valid, if balance can always restore the balance condition after one insert, delete or join operation. It would be best to fully characterize valid combination of parameters, but it difficult to do so. The reason is that the parameter validity heavily relies on the fact that small trees still have integral sizes – if the only available counterexamples are non-integral, the balance is always restored. However, because of this behaviour it is difficult to give generic characterization of parameter validity. We therefore rule out parameters which are definitely not valid and then prove the validity only for several chosen parameters. It is easy to see that $\omega \geq 5$ and $\omega = 2$ are not valid for any $\alpha$ in the sense of the original balance condition, i.e., with $\delta = 0$: In the situation in Fig. 2 neither single nor double rotation can restore balance.

To get more accurate idea, we evaluated validity of parameters on all trees up to size of 1 million – the results are displayed in Fig. 3. The code used to generate this figure is listed in Appendix A. When choosing the parameters, the value of $\omega$ is the most important, because it defines the height of the tree. On
Fig. 2. Parameters $\omega = 2$ and $\omega \geq 5$ are not valid for any $\alpha$ and $\delta = 0$.

Fig. 3. The space of parameter $(\omega, \alpha, \delta)$. The values of $\omega$ and $\alpha$ are displayed on the $x$ and $y$ axis, respectively. Every dashed square consists of four smaller squares, which correspond to the $\delta$ values $\{0, 1\}$. Black denotes non-valid parameters, white denotes parameters which are valid for trees of size up to 1 million. For example, when $\omega = 4$ and $\alpha = 2$, $\delta \in \{0, 3\}$ is valid and $\delta \in \{1, 2\}$ is not valid.
the other hand, the value of \( \alpha \) is quite unimportant – it affects only the internal implementation of balance. The value of \( \delta \) is kept as low as possible, since it increases imbalance of tree sizes.

After inspection of Fig. 3 we have chosen integer parameters \((\omega = 3, \alpha = 2, \delta = 0)\) and \((\omega = 4, \alpha = 2, \delta = 0)\) and also parameters \((\omega = 2.5, \alpha = 1.5, \delta = 1)\), where the value of \( \omega \) is the smallest possible. The last parameters are not integral, but we can perform multiplication by \( \omega \) or \( \alpha \) using right bit shift.

5.1 Validity of \( w = 2.5, w = 3 \) and \( w = 4 \)

We now prove the validity of chosen parameters \((\omega = 2.5, \alpha = 1.5, \delta = 1)\), \((\omega = 3, \alpha = 2, \delta = 0)\) and \((\omega = 4, \alpha = 2, \delta = 0)\). Because the values of \( \alpha \) and \( \delta \) are determined by \( \omega \), we identify these sets of parameters only by the value of \( \omega \).

Consider performing balance after the balance is lost. Without loss of generality assume the right subtree is the bigger one and denote \( n \) and \( m \) the sizes of the left and right subtrees, respectively. We will use the notation of the tree size and the tree itself interchangeably.

Because the balance is lost, we have now \( \omega n + \delta < m \). The insert operation causes imbalance by exactly one element, so it is never worse than imbalance caused by a delete operation. Therefore we have to consider only two possibilities how the imbalance was caused – delete or join operation. If the last operation was delete, we know that \( \omega n + \delta \geq m - \omega \). If the last operation was join with the subtree of size \( z \), we know that \( \omega n + \delta \geq m - z \).

During the join operation the tree \( z \) was small enough to be recursively joined with subtree \( m \), so we have \( \omega z + \delta < n + 1 + (m - z) \), so \( z < \frac{n + 1 + m - \delta}{\omega + 1} \) and therefore \( m - \frac{n + m + 1 - \delta}{\omega + 1} < \omega n + \delta \). 

\[
m < \frac{\omega + 1}{\omega} \left( \omega n + \delta + \frac{n + \omega \delta + 1}{\omega + 1} \right),
\]

\[
m < \left( \omega + 1 + \frac{1}{\omega} \right) n + \delta + \frac{1}{\omega}.
\]

To summarize:

\[
m > \omega n + \delta, \quad m - \omega \leq \omega n + \delta, \quad m < \left( \omega + 1 + \frac{1}{\omega} \right) n + \delta + \frac{1}{\omega}.
\]

5.2 Correctness of a single rotation

Let \( x \) and \( y \) denote the subtrees of the tree \( m \). We perform a single rotation iff \( x < \alpha y \) and in that case we have the following inequalities:

\[
\omega x + \delta \geq y \Rightarrow (\omega + 1)x + \delta \geq m - 1,
\]

\[
x < \alpha y \Rightarrow x < \frac{\alpha}{\alpha + 1}(m - 1), \quad y > \frac{1}{\alpha + 1}(m - 1).
\]

At first we need to solve the cases where \( n, x \) or \( y \) are zero, as the balance condition is different in that case. All such cases are shown in Fig. 4.

In the case when all subtrees are nonempty, we need to validate the balance condition in each of the two new trees:
Fig. 4. Cases when \( n, x \text{ or } y \) are zero and a single rotation is performed.

\(-\omega n + \delta \geq x \) after delete: \( x < \frac{\alpha}{\alpha+1}(m-1) \leq \frac{\alpha}{\alpha+1}(\omega n + \delta + \omega - 1)\)

\(-\omega n + \delta \geq x \) after join: \( x < \frac{\alpha}{\alpha+1}(m-1) \leq \frac{\alpha}{\alpha+1}((\omega + 1 + \frac{1}{\omega})n + \delta + \frac{1}{\omega} - 1)\)

\(-\omega x + \delta \geq n \): \( n < \frac{m-\delta}{\omega} \leq \frac{\omega+1}{\omega}x + \frac{1}{\omega}\)

\(-\omega n + \delta \geq x \) after delete: \( x < \frac{\alpha}{\alpha+1}(m-1) \leq \frac{\alpha}{\alpha+1}(\omega n + \delta + \omega - 1)\)

\(-\omega n + \delta \geq x \) after join: \( x < \frac{\alpha}{\alpha+1}(m-1) \leq \frac{\alpha}{\alpha+1}((\omega + 1 + \frac{1}{\omega})n + \delta + \frac{1}{\omega} - 1)\)

\(-\omega x + \delta \geq n \): \( n < \frac{m-\delta}{\omega} \leq \frac{\omega+1}{\omega}x + \frac{1}{\omega}\)

\(-\omega y + \delta \geq n + 1 + x: n + 1 + x = n + m - y \leq \frac{m-1}{\omega} + m - y = m\frac{\omega+1}{\omega} - y - \frac{1}{\omega} \leq \frac{(\alpha + 1)y + 1}{\omega}\)

\((-\omega y + \delta \geq n + 1 + x: n + 1 + x = n + m - y \leq \frac{m-1}{\omega} + m - y = m\frac{\omega+1}{\omega} - y - \frac{1}{\omega} \leq \frac{(\alpha + 1)y + 1}{\omega}\)

The third and the fourth inequalities obviously hold. To see that also the first, second and fifth inequalities hold, we evaluate the resulting inequalities and use the fact that the tree sizes are positive integers:

<table>
<thead>
<tr>
<th>( \omega )</th>
<th>( \omega n + \delta \geq x ) after delete</th>
<th>( \omega n + \delta \geq x ) after join</th>
<th>( \omega y + \delta \geq n + 1 + x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>( x &lt; \frac{3}{2}n + \frac{3}{7} )</td>
<td>( x &lt; \frac{17}{25}n + \frac{6}{25} )</td>
<td>( n + 1 + x &lt; \frac{5}{2}y + 1 )</td>
</tr>
<tr>
<td>3</td>
<td>( x &lt; \frac{2}{3}n + \frac{1}{3} )</td>
<td>( x &lt; \frac{11}{25}n + \frac{6}{25} )</td>
<td>( n + 1 + x &lt; 3y + 1 )</td>
</tr>
<tr>
<td>4</td>
<td>( x &lt; \frac{8}{3}n + 2 )</td>
<td>( x &lt; \frac{7}{2}n - \frac{1}{2} )</td>
<td>( n + 1 + x &lt; \frac{11}{4}y + 1 )</td>
</tr>
</tbody>
</table>

The linear coefficients are always less or equal the required ones and it is simple to verify that all inequalities hold also for small integer sizes.

5.3 Correctness of a double rotation

When performing a double rotation, we have the following inequalities:

\( \text{any child } a \text{ of } b \Rightarrow (\omega + 1)a + \delta \geq b - 1 \),

\( \text{any child } a \text{ of } b \Rightarrow (\omega + 1)a \leq \omega(b - 1) + \delta \),

\( x \geq \alpha y \Rightarrow x \geq \frac{\alpha}{\alpha+1}(m-1), \ y \leq \frac{1}{\alpha+1}(m-1) \).

Once again we need to solve the cases when \( n, y, s \text{ or } t \) are zero – we enumerate these cases in Fig. 5.

When all subtrees are nonempty we create three new trees, so we have to check six inequalities:
Fig. 5. Cases when \( n, y, s \) or \( t \) are zero and a double rotation is performed.

- \( \omega n + \delta \geq s \) after delete: \( s \leq \frac{\omega}{\omega+1}(x-1+\frac{\delta}{\omega}) \leq \frac{\omega}{\omega+1}(\frac{\omega}{\omega+1}(m-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega} \)
- \( \omega n + \delta \geq s \) after join: \( s \leq \frac{\omega}{\omega+1}(x-1+\frac{\delta}{\omega}) \leq \frac{\omega}{\omega+1}(\frac{\omega}{\omega+1}(m-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega})-1+\frac{\delta}{\omega} \)

All but the first three inequalities obviously hold for positive integral sizes. In order to prove that the first three inequalities hold, we again evaluate the resulting inequalities and use the fact that the sizes are positive integers:

\[
\begin{align*}
\omega n + \delta &\geq s \text{ after delete: } \omega(n+1+s) + \delta \geq \omega(n+1) + t \geq m - \delta + t \geq x - \delta + 1 + y + t \\
\omega(n+1+s) + \delta &\geq t+1+y \text{ after join: } t+1+y \leq \omega s + \delta + 1 + y \leq \omega s + \delta + 1 + \frac{m-1}{\alpha+1} \leq \omega s + \delta + 1 + \frac{\omega+1}{\omega+1}(\frac{1+\frac{1}{\alpha}}{\omega+1}) n + 1 + \omega(\delta-1)n+1 + \omega(s+\delta) + m-1 \leq \omega t + \delta + 1 + \frac{\omega+1}{\omega+1}(\omega+1)y + \delta + \frac{1+\frac{1}{\alpha}}{\omega+1} \leq \omega t + \frac{\omega+\delta+1}{\omega+1} + \frac{\omega+1}{\omega+1} y + \delta
\end{align*}
\]

The linear coefficients are less or equal than the required ones and for small positive integral sizes the resulting inequalities imply the required ones, which concludes the proof.
6 BB-ω trees height

If the balance condition holds and $\delta \leq 1$, we know that the size of a tree decreases by at least a factor of $\frac{\omega}{\omega+1}$. Therefore the maximum height of a tree is $\log_2 n$. But this is merely an upper bound – it is frequently not possible for the balance condition to be tight, because the tree sizes are integers.

To get an accurate estimate, we compute the maximum heights of BB-ω trees up to size of 1 million. We used the following recursive definition:

```haskell
-- Returns the list [ max height of BB-w tree with n elements | n <- [1..] ].
heights :: Ratio Int -> Int -> [Int]
heights w d = result
  where
    result = 1 : 2 : compute_heights 3 1 result
    compute_heights n r rhs@(rhs_head : rhs_tail)
      | w*(n-1-(r+1))%1 + d%1 >= (r+1)%1 = compute_heights n (r+1) rhs_tail
      | otherwise = 1 + rhs_head : compute_heights (n+1) r rhs
```

The function `compute_heights` gets the size of the tree $n$, the size of its right subtree $r$ and also a list of maximum heights of BB-ω trees of $r$ and more elements. It constructs the highest tree of size $n$ by using the largest possible right subtree, and then using the highest tree of such size.

The resulting heights are presented in Fig. 6. The heights are divided by $\lceil \log_2 n \rceil$, so the optimal height is 1. Notice that the height of a BB-2.5 tree is always smaller than 2 for less than million elements – such height is better than the height of a red-black tree of the same size.

<table>
<thead>
<tr>
<th>size of BB-ω tree</th>
<th>height divided by $\lceil \log_2 n \rceil$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\omega = 2.5$</td>
</tr>
<tr>
<td>10</td>
<td>1.33</td>
</tr>
<tr>
<td>100</td>
<td>1.57</td>
</tr>
<tr>
<td>1000</td>
<td>1.70</td>
</tr>
<tr>
<td>10000</td>
<td>1.84</td>
</tr>
<tr>
<td>100000</td>
<td>1.86</td>
</tr>
<tr>
<td>1000000</td>
<td>1.90</td>
</tr>
<tr>
<td>upper bound</td>
<td>2.06</td>
</tr>
</tbody>
</table>

Fig. 6. Maximum heights of BB-ω trees with $\omega = 2.5$, $\omega = 3$ and $\omega = 4$.

7 The performance of BB-2.5, BB-3 and BB-4 trees

With various possible $\omega$ to use, a search for the optimum value is in order. Is some value of $\omega$ universally the best one or does different usage patterns call for specific $\omega$ values?
We know that smaller values of $\omega$ result in lower trees. That seems advantageous, because the time complexity of many operations is proportional to the tree height.

In order to compare different values of $\omega$, we measured the number of invocations of \texttt{balance} function. We inserted and then deleted $10^{(1..6)}$ elements, in both ascending and uniformly random order, and measured the number of invocations of \texttt{balance} during each phase. The results are displayed in Fig. 7.

<table>
<thead>
<tr>
<th></th>
<th>$w = 2.5$</th>
<th>$w = 3.0$</th>
<th>$w = 4.0$</th>
<th>$w = 2.5$</th>
<th>$w = 3.0$</th>
<th>$w = 4.0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>consecutive 10 elements</td>
<td>25</td>
<td>25</td>
<td>26</td>
<td>11</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>random 10 elements</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>consecutive $10^2$ elements</td>
<td>617</td>
<td>657</td>
<td>769</td>
<td>362</td>
<td>349</td>
<td>302</td>
</tr>
<tr>
<td>random $10^2$ elements</td>
<td>542</td>
<td>549</td>
<td>562</td>
<td>377</td>
<td>376</td>
<td>413</td>
</tr>
<tr>
<td>consecutive $10^3$ elements</td>
<td>10245</td>
<td>11439</td>
<td>13997</td>
<td>6554</td>
<td>6116</td>
<td>5500</td>
</tr>
<tr>
<td>random $10^3$ elements</td>
<td>8700</td>
<td>8753</td>
<td>8953</td>
<td>7162</td>
<td>7177</td>
<td>7377</td>
</tr>
<tr>
<td>consecutive $10^4$ elements</td>
<td>143685</td>
<td>163261</td>
<td>206406</td>
<td>94865</td>
<td>88487</td>
<td>79938</td>
</tr>
<tr>
<td>random $10^4$ elements</td>
<td>121192</td>
<td>121623</td>
<td>124204</td>
<td>105251</td>
<td>105854</td>
<td>108362</td>
</tr>
<tr>
<td>consecutive $10^5$ elements</td>
<td>1852582</td>
<td>2133997</td>
<td>2722419</td>
<td>1251621</td>
<td>1175569</td>
<td>1042398</td>
</tr>
<tr>
<td>random $10^5$ elements</td>
<td>1554230</td>
<td>1562168</td>
<td>1595269</td>
<td>1395871</td>
<td>1402939</td>
<td>1434371</td>
</tr>
<tr>
<td>consecutive $10^6$ elements</td>
<td>22701321</td>
<td>26336469</td>
<td>33878677</td>
<td>15492747</td>
<td>14429384</td>
<td>12974950</td>
</tr>
<tr>
<td>random $10^6$ elements</td>
<td>18956075</td>
<td>19074599</td>
<td>19476673</td>
<td>17367930</td>
<td>17480730</td>
<td>17856278</td>
</tr>
</tbody>
</table>

**Fig. 7.** The number of \texttt{balance} calls during inserting and deleting elements.

In case of ascending elements, smaller $\omega$ values perform better during insertion – the difference between $\omega = 2.5$ and $\omega = 4$ is nearly 50% for large number of elements. On the other hand, higher $\omega$ values perform better during deletion, although the difference is only 18% at most. In case of random elements, lower values of $\omega$ are always better, but the difference is less noticeable in this case.

We also performed the benchmark of running time of \texttt{insert}, \texttt{lookup} and \texttt{delete} operations. We used the \texttt{criterion} package [11], a commonly used Haskell benchmarking framework. All benchmarks were performed on a dedicated machine with Intel Xeon processor and 4GB RAM, using 32-bit GHC 7.0.1. The benchmarking process works by calling the benchmarked method on given input data and forcing the evaluation of the result. Because the benchmarked method can take only microseconds to execute, the benchmarking framework repeats the execution of the method until it takes reasonable time (imagine 50ms) and then divides the elapsed time by the number of iterations. This process is repeated 100 times to get the whole distribution of the time needed, and the mean and the confidence interval are produced.

The benchmarks are similar to our previous experiment – we insert, locate and delete $10^{(1..6)}$ elements of type \texttt{Int}, which are both in ascending and uniformly random order. We used the implementation of \texttt{balance} from the \texttt{containers} package – we already improved this implementation in [9]. The re-
sulting execution times are normalised with respect to one of the implementations and presented as percentages. The overview is in Fig. 8. (Ignore the trees with One subscript, they are explained in the next section.) Here the geometric mean of running times for all input sizes $10^3$ to $10^6$ is displayed. The detailed results and the benchmark itself are available on the author’s website http://fox.ucw.cz/papers/bbtree.

![Fig. 8. The normalized execution times of BB-$\omega$ trees with various $\omega$.](image-url)

The findings are similar to the previous experiment – if the elements are in random order, the value of $\omega$ makes little difference, and smaller values perform slightly better. In case of ascending elements, smaller $\omega$ are better when inserting and larger when deleting. As expected, the lookup operation runs faster for smaller values of $\omega$, independently on the order of elements.
The proposed representation of a BB-ω tree provides room for improvements in terms of memory efficiency – if the tree contains $n$ nodes, there are $n + 1$ Nil constructors in the whole tree, because every Node constructors contains two sub-trees. We can improve the situation by introducing additional data constructor for a tree of size one:

```hs
data BBTree a = Nil            -- empty tree
  | One a               -- tree of size one
  | Node
    (BBTree a)      -- left subtree
    Int             -- size of this tree
    a               -- element stored in the node
    (BBTree a)      -- right subtree
```

Leaves are represented efficiently with this data-type. However, the trees of size 2 still require one Nil constructor.

To determine the benefit of the new data constructor we need to bound the number of Nil constructors in the tree. A Nil constructor appears in a tree of size 2 and if there are $t$ trees of size 2, there need to be at least $(t - 1)$ internal Nodes for these $t$ trees to be reachable from the root. Therefore, there can be at most $n/3$ Nil constructors in the tree. This implies that the number of One constructors is between $n/3$ and $n/2$. Experimental measurements presented in Fig. 9 show that a tree created by repeatedly inserting ascending elements contains $n/2$ One and no Nil constructors, and a tree created by inserting uniformly random elements contains approximately $0.43n$ One and $0.14n$ Nil constructors.

<table>
<thead>
<tr>
<th>$T_{\text{On2.5}}$</th>
<th>$T_{\text{On3.0}}$</th>
<th>$T_{\text{On4.0}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>random 10 elements</td>
<td>50.0% 50.0% 50.0%</td>
<td>50.0% 50.0% 50.0%</td>
</tr>
<tr>
<td>random 10^2 elements</td>
<td>45.5% 45.5% 45.5%</td>
<td>43.6% 43.6% 43.6%</td>
</tr>
<tr>
<td>random 10^3 elements</td>
<td>43.0% 43.0% 42.8%</td>
<td>43.0% 43.0% 42.8%</td>
</tr>
<tr>
<td>random 10^4 elements</td>
<td>42.8% 42.8% 42.9%</td>
<td>42.8% 42.8% 42.9%</td>
</tr>
<tr>
<td>random 10^5 elements</td>
<td>42.9% 42.9% 42.9%</td>
<td>42.9% 42.9% 42.9%</td>
</tr>
<tr>
<td>random 10^6 elements</td>
<td>42.9% 42.9% 42.9%</td>
<td>42.9% 42.9% 42.9%</td>
</tr>
</tbody>
</table>

Fig. 9. The percentage of One constructors in a BB-ω tree.

Considering the memory representation used by the GHC compiler, the Node constructor occupies 5 words and One constructor occupies 2 words, so the new representation takes 20-30% less memory. The time complexity of the new representation is also better as shown in Fig. 8. Especially note the speedup of the fold operation, which is the result of decreased number of Nil constructors in
the tree. The only disadvantage is the increase of the code size – but this affects
the library author only.

We could also add a fourth data constructor to represent a tree of size 2. That would result in no \texttt{Nil} constructors in a nonempty tree. The disadvantage is
further code size increase and also a noticeable time penalty – on 32bit machines
GHC uses pointer tagging to distinguish data constructors without the pointer
dereference, which is described in detail in \cite{7}. This technique works with types
with at most three data constructors (and up to 7 different constructors on 64bit
machines), so it is not advantageous to add a fourth data constructor.

8.1 The order of data constructors

When implementing the data-type with the \texttt{One} constructor, we found out that
the order of data constructors in the definition of the data-type notably af-
fects the performance. On Fig. 10 you can see the time improvements in the
benchmark from the previous section, when we reordered the constructors to
the following order: \texttt{Node} first, then \texttt{One} and \texttt{Nil} last.

\begin{tabular}{l|ccc}
 & $T_{\text{One}2.5}$ & $T_{\text{One}3.0}$ & $T_{\text{One}4.0}$ \\
\hline
insert\_asc & 5.1\% & 6.8\% & 6.6\% \\
insert\_rnd & 4.5\% & 5.2\% & 5.0\% \\
llookup\_asc & 7.4\% & 6.1\% & 6.2\% \\
llookup\_rnd & 6.1\% & 5.4\% & 5.4\% \\
delete\_asc & 5.3\% & 8.4\% & 8.5\% \\
delete\_rnd & 4.4\% & 4.8\% & 5.0\% \\
fold\_asc & 8.9\% & 9.5\% & 13.1\% \\
fold\_rnd & 10.1\% & 10.5\% & 9.4\% \\
\hline
\end{tabular}

\textbf{Fig. 10.} The improvements of time complexity after reordering the data constructors.

We believe the reason for the performance improvement is the following:
When matching data constructors, a conditional forward jump is made if the
constructor is not the first one from the data-type definition. Then another
conditional forward jump is made if the constructor is not the second one from
the data-type definition. In other words, it takes $i-1$ conditional forward jumps
to match the $i$-th constructor from the data-type definition, and these forward
jumps are usually mispredicted (forward jumps are expected not to be taken).
It is therefore most efficient to list the data constructor in decreasing order of
their frequency.

9 Conclusions

We described balanced trees and explicitly proved their correctness for several
representative parameter combinations. For these parameters we also measured
their runtime performance. The resulting implementation is comparable to other available on Hackage (this work started already in [9]). We also focused on memory complexity and improved it by changing the data-type representation. During this process we discovered the effect of the data constructors order in the data-type definition on the performance.

Several goals remain for future work. In our further efforts, we will incorporate the improvements described here in the containers package. We will also benchmark the effect of reordering data constructors of other data structures from the containers package – especially the IntMap, IntSet, HashMap and HashSet, which all use three data constructors. Also the benchmark of BB-ω trees could be extended to include set operations like union, intersection and others. We already described a benchmark with a union operation in [9].

9.1 Related work

The original weight balanced trees were described in [8], with two parameters with values 1+√2 and √2. Because these are not integers, the resulting algorithm is not very practical. Adams created a variant of balanced trees, the BB-ω trees, and described them in papers [1] and [2]. Unfortunately, the proof is erroneous – the paper concludes that for α = 2 the valid parameters are ω ≥ 4.646.

The error in the proof was known by several people, but in 2010 a bug was also found in the Haskell implementation – in the Data.Set and Data.Map modules from the containers package. The recent paper [5] deals with the correctness of the original weight balanced trees (equivalent to setting δ = ω − 1 in our definition) and proves in Coq, that for δ = ω − 1 the only integral valid parameters are ω = 3 and α = 2. Our proof on the other hand is explicit, and proves validity of only some chosen parameters. It covers both the original weighted trees and Adams’ trees.

References

A Generating the Fig. 3

When generating the Fig. 3 of valid parameters for all trees up to size of 1 million, we used the following code:

```haskell
max_n = 1000000
find_min x p | p x = last $ x : takeWhile p [x-1, x-2 .. 0]
| otherwise = head $ dropWhile (not . p) [x+1, x+2 ..]

test w a d = and [delete n m && join n m | n <- [0 .. max_n],
let m = flr $ max 1 (w * n + d)]

where
delete n m = n == 0 || rebalance (n-1) m
join n m = rebalance n (m+increment)
where increment = max 1 $ ceil ((n+m+1-d) / (w+1) - 1)

rebalance n m = and [rebalance' n m x | x <- nub [x_min, x_mid - 1,
x_mid, m - 1 - x_min]]
where x_min = find_min (flr $ m / (w+1)) (\x -> balanced x (m-1-x))
x_mid = find_min (flr $ m * a / (a+1)) (\x -> x >= a * (m-1-x))

rebalance' n m x
| x < a * y = balanced n x && balanced (n + 1 + x) y
| otherwise = balanced n s && balanced t y && balanced (n+1+s) (t+1+y) &&
balanced n t && balanced s y && balanced (n+1+t) (s+1+y)
where (y, s, t) = (m-1-x, find_min (flr$x/(w+1)) (\s->balanced s (x-1-s)), x-1-s)

balanced n m = max 1 (w * n + d) >= m && n <= max 1 (w * m + d)

flr, ceil :: Double -> Double
flr = fromInteger . floor
ceil = fromInteger . ceiling

results = [(w, a, d, test w a d) | w <- [2, 2.125 .. 5],
a <- [1, 1.125 .. 3], d <- [0 .. 3]]
```

It relies on the fact that when there is a tree which cannot be balanced, there also exists a counterexample with a subtree as large as the balance condition allows. Therefore, for a fixed value of \( n \) it is enough to try the largest possible \( m \) and for a fixed value of \( m \) it is enough to verify that the balance condition is restored when considering the smallest and the largest subtree causing a single rotation and the smallest and the largest subtree causing a double rotation.
The Time of Space Invaders Will Come to Pass
A CS1 Functional Video Game Journey from Structural
Recursion to Generative and Accumulative Recursion

Marco T. Morazán
Seton Hall University, South Orange, NJ, USA
morazanm@shu.edu

Abstract. The use of video games to teach introduction courses to programming and Computer Science is a trend that is currently flourishing. One of the most successful and promising approaches uses functional video games to get students interested and engaged in programming. This approach is successful, in part, because functional video games provide a domain of interest to most Computer Science undergraduates and remove the need to reason about designing state-based programs. A plethora of examples exist that have students develop games exploiting structural recursion which resemble such classics as Space Invaders and Snake. Once students master the basics of structural recursion the time comes to move beyond structural recursion to other forms of recursion such as generative and accumulative recursion. It is up to the instructor to harness the enthusiasm and appetite that students have to develop more video games. This requires finding games that require the generation of subproblems in the same class as the input problem or that require accumulators to be successfully played or solved. This article presents a road map to make the transition from structural recursion to accumulative recursion using the N-puzzle problem as motivation to capture student enthusiasm and exploit what they have learned about program design. The N-Puzzle was also chosen to demonstrate that informed heuristic search strategies, traditionally the domain of undergraduate courses in Artificial Intelligence, are within the grasp of CS1 students. CS1 students can reason such an algorithm into existence instead of simply studying one in a textbook. If the work described in this article is replicated elsewhere, there is no doubt that it will be an exciting time for Computer Science education and it will elevate the relevance of functional programming in the minds of future CS professionals.

1 Introduction

Based on the teaching philosophy of program by design (PBD) put forth in the textbook How to Design Programs (HtDP) [2], the use of functional video games to teach introduction courses to programming and Computer Science is a trend that is currently flourishing. At the heart of the PBD philosophy is what is coined as the design recipe—a series of steps that students can follow to write programs. These steps include the development of data definitions based on data
analysis, the development of contracts and function headers, the development of function templates for all data definitions, the specializing of function templates to create functions, and the development and running of tests. At the beginning of an introduction course, the focus is on solving problems using structural recursion. One of the most successful and promising implementation approaches to a PBD-based course uses functional video games to get students interested and engaged in programming. This approach is successful, in part, because functional video games provide a domain of interest to most Computer Science undergraduates and remove the need to reason about designing state-based programs. A plethora of examples exist that have students develop games exploiting structural recursion which resemble such classics as Space Invaders and Snake [1, 5].

Once students master the basics of structural recursion the time comes to move beyond structural recursion to other forms of recursion such as generative and accumulative recursion. It is up to the instructor to harness the enthusiasm and appetite that students have to develop more video games to motivate these topics. This requires identifying games that can not be played or solved by only using structural recursion. It is important to note, however, that the goal is not to make students masters at developing video games. Instead, the goal is to make students interested in generative and accumulative recursion by showing them how they are needed and/or used in a video game. Surprisingly, there are not many examples in an HtDP-based curriculum of video games that require students to go beyond structural recursion.

This article advocates the position that video games ought to be used to motivate the need to study generative and accumulative recursion in the CS1 classroom. It presents a road map to make the transition from structural recursion to generative and accumulative recursion using the N-puzzle problem as motivation to capture student enthusiasm. The primary goal is to introduce these topics while at the same time exploiting and reinforcing what students have learned about program by design and structural recursion. Secondary goals are to expose students to ideas that they may encounter in upper-level courses such as heuristics in an Artificial Intelligence course and the use of random number generators. The article is organized as follows. Section 2 briefly describes the N-puzzle game. Section 3 describes the first encounter of students with the N-puzzle game in the classroom and discusses opportunities the game presents to reinforce the lessons of program by design using structural recursion. Section 4 discusses an initial strategy to finding a solution leading to the need for generative recursion. Section 5 discusses how the need for accumulators arises and how accumulative recursion is used in the N-puzzle game. Section 7 discusses related work and Section 8 draws some conclusions and briefly outlines future work.

2 The N-Puzzle Game

The N-puzzle game is one that is likely to be familiar to an international milieu of students and is simple enough that students can easily grasp how the game
works. It consists of an $N \times N$ board with $N^2 - 1$ tiles\(^1\) and an empty square or
blank space that does not contain a tile. Each tile contains some form of symbolic
or numeric data. Figure 1 displays a sample board using numeric tiles for the
3-puzzle game in which the empty space is at the center of the board. Every
N-puzzle game must also define a winning board. That is, a board that defines
the solution to the puzzle. Figure 2 displays the traditional winning board for
the numeric 3-puzzle problem.

A player can move tiles by swapping the blank space with one of its neighbors.
The goal of the game is to make a sequence of moves that lead to the winning
board. A player, of course, at some point during the game may feel stuck and
the game should provide a mechanism, like a help button, to ask the computer
to make the next move. The help button, of course, requires the program to first
solve the puzzle before making a move towards the solution on behalf of the
player.

To make the game more challenging and more interesting the game can be
parameterized with a constant $N$. In this manner, students are free to make the
board larger or smaller according to the level of the challenge they desire.

3 The First Encounter with the N-Puzzle Game in CS1

Students that are presented with the N-puzzle game have gone through the first
four parts of HtDP that cover program by design with structures, structurally
recursive data types, and abstraction. They have experience designing programs
that process, for example, lists and trees as well as familiarity with basic ab-
straction patterns that involve the use of higher-order functions such as map
and filter.

When students are first presented with the N-puzzle game, they are asked
what is changing while the game is played and how it can be represented. This

\(^1\) The choice of a square board is arbitrary, but facilitates developing a program
A board is either:
1. empty
2. (cons number b), where b is a board

Template for functions on boards:
(define (f-on-board a-board)
  (cond [(empty? a-board) ...]
        [else ...(first a-board)...(rest a-board)]))

Fig. 3. Data definition for boards and a template for functions on boards.

leads to defining a board as a list of numbers and to a template for functions on boards both of which are displayed in Figure 3. This brings the N-puzzle game into a realm that is familiar to the students and provides an opportunity to reinforce lessons on structural recursion.

To get students started, the first tasks they are asked to solve can be done using structural recursion and/or abstraction such as building the representation of the winning board, finding the position of the empty space, and swapping two tiles (eventually used to make moves). The solutions presented may vary with some students defining such functions using structural recursion and some students using higher-order functions. Typical solutions for the game with numeric tiles are displayed in Figure 4.

The initial encounter with the N-puzzle game also provides an opportunity to perform data analysis that leads to the realization that more than structural recursion is required to implement the help button. Students are asked what does it mean to find a solution when the player requests the computer to make the next move. After some discussion, it becomes clear that finding a solution is finding a sequence of moves from the current board to the winning board. Students, in general, can grasp without too much trouble the idea that finding such a sequence of moves for board \( b \) means finding a solution for one of the possible successors of \( b \), \( child_b \), obtained by making a single move and adding the move that takes \( b \) to \( child_b \). The question then becomes which move will be chosen to generate the child of \( b \) that is to be explored.

The problem, of course, is that such a strategy is no longer structural recursion. Structural recursion guarantees that the size of the subproblems (i.e., finding a solution starting from some \( child_b \)) are smaller than the problem of finding a solution to the original problem (i.e., finding a solution starting at \( b \)) and are derived from the structure of the input. This is not the case, because in general some sequences starting at \( b \) are infinite as are sequences starting at any \( child_b \) and \( child_b \) is not used to build \( b \). The question then becomes how do you solve problems that generate subproblems that are not guaranteed to be smaller than the original problem and are not part of the structure of the

---

2 Some students suggest the tiles are pieces of an image, but most students feel more comfortable with numeric tiles.
4 Finding a Solution

After using the N-puzzle game to discover the need for generative recursion, students are given several examples on how to design programs based on generative recursion. Examples outlined in HtDP include quicksort, fractals, binary search, Newton’s method, and backtracking algorithms such as traversing a graph to find a path from node A to node C. Of these, the most relevant to finding a solution to an N-puzzle are the backtracking algorithms, because traversing a graph with cycles can lead to a path of infinite length precisely in the same manner that some sequences of moves are infinite in the N-puzzle problem. HtDP presents a solution to find a path from node A to node C in an acyclic graph using a depth-first traversal and postpones finding the solution for a graph with cycles to motivate accumulative recursion.

In the N-puzzle problem, of course, we are for the most part unable to restrict our sequences of moves to those that are finite. This presents the opportunity to introduce beginning students to heuristics. A heuristic can be used to choose which child of b is chosen to explore for a solution. It is important to remark to students that a heuristic is rule that estimates how many moves away from the current board is from the winning board. A heuristic is used hoping it will lead to a solution. At this point, most students will have no way to judge this statement and simply trust the professor. This trust opens the door for communicating
Fig. 5. Code for computing the Manhattan distance of a board

lessons on the importance of testing and careful design in programming. As the reader knows, this approach is destined to immediate failure, but also to triumph after the process of iterative refinement is started.

There is a simple heuristic students can understand and implement for the N-puzzle problem. The heuristic chooses to explore the child of \( b \) that has the smallest Manhattan distance. The Manhattan distance of a board is the sum of how far away each tile is from its correct position. For example, the Manhattan distance of the board in Figure 2 is 0 given that all tiles are in the correct position. In Figure 1, tile 1 is in the right position and contributes 0 to the Manhattan distance while the blank space, in position 4 and whose correct position is 8, contributes 2 to the Manhattan distance. The code to compute the Manhattan distance of a board is displayed in Figure 5. Observe that the code only requires arithmetic and structural recursion on natural numbers which provides the opportunity to reinforce material students have already seen and to make the material relevant to their interests in video games.

Armed with the power of a heuristic, students can now delve into designing an N-puzzle solver. The basic idea is that given a board their program needs to return a non-empty list of boards, called a sequence, that contains all the boards in the sequence of moves from the given board to the winning board. These ideas lead quite naturally to the design of a depth-first search algorithm. If the given board is the same as the winning board, then the solution is trivial: a list con-
(define (find-solution-dfs b)
  (cond [(equal? b WIN) (list b)]
        [else
         (local [(define children (generate-children b))]
                (cons b (find-solution-dfs (best-child children))))]))

(define (generate-children b)
  (local [(define blank-pos (get-blank-sq-num b))]
         (map (lambda (p)
               (swap-tiles b blank-pos p))
               (blank-neighs blank-pos))))

(define (best-child lob)
  (cond [(empty? (rest lob)) (car lob)]
        [else
         (local [(define best-of-rest (best-child (rest lob)))]
                (cond [(< (manhattan-distance (car lob))
                       (manhattan-distance best-of-rest))
                       (car lob)]
                       [else best-of-rest]))]))

Fig. 6. Code for depth-first search for a solution without backtracking.

The benefits of using the N-puzzle to reinforce lessons from structural recursion to motivate generative recursion and to capture the interest of students are likely to be self-evident to any instructor at the helm of a CS1 class. Clearly, this video game also provides the opportunity to introduce CS1 students quite naturally to depth-first search and to heuristics-based programming which is quite uncommon as far as the author knows. There are, however, two more benefits that deserve to be mentioned. These are reinforcing the value of testing and the value of iterative refinement. The code in Figure 6 does, indeed, find a solution for some boards. Testing, however, reveals that it fails to return a solution for some boards. This leads to an exploration of why the program, which seems
Fig. 7. Illustration of why a depth-first path (dashed) does not lead to a solution.

quite reasonable to most students, fails to return a solution for some boards and how it can be improved to guarantee that a solution is always returned (for a legal board).

5 The Need to Remember Leads to Accumulators

The exploration of why the program fails to return a solution to some boards leads to the discussion of a situation like the one depicted in Figure 7. If the current board is the one in the root of the tree, it has two children both of which have a Manhattan distance of 18. The algorithm chooses the right child as the board to explore. This board has three children that, from left to right, have Manhattan distances of 20, 20, and 16. The rightmost child is chosen for exploration as it has the smallest Manhattan distance. At this point, all students can see the problem. The algorithm cycles through the same set of boards never choosing a different board to escape the cycle. In other words, students understand why there is an infinite recursion and why it is impossible to argue that the algorithm terminates as is required by the design recipe for programs based on generative recursion.

At this point, students are guided to think that a solution to this problem requires that all sequences starting at the given board must be explored instead of choosing to only explore the sequence of best children. This requires that all paths explored so far be remembered. Through this analysis, students have entered the realm of accumulative recursion and this is used as motivation to return to HtDP and study how to design programs that exploit this new kind of recursion.

One of the functions students can develop while exploring how to design programs that use accumulative recursion is a function to present a player with
an initial board to solve in the N-puzzle problem. This presents an interesting
task, because not all possible orderings of tiles in a board are valid boards in
the N-puzzle game. In the 3-puzzle game, for example, the ordering that has 1,
2, 3 in the first row, 4, 5, 6, in the second row, and 8, 7, 0 in the third row is
an invalid board. The challenge, therefore, is to design a strategy to compute
an initial board that does not simply randomly assign tiles to positions in the
board. After some discussion, a natural strategy to follow is to start from the
winning board and randomly make \( k \) valid moves. This strategy is a good one
to choose in CS1 for three reasons. The first is that it provides students at this
early stage in their studies with an example of where the use of randomness is
useful. The second is that it requires an accumulator to “remember” the board
created so far. That is, after every random move a new board is created and
the new board needs to be used to make any further moves. The third is that it
brings accumulative recursion into the domain of structural recursion on natural
numbers—a familiar world for students that have followed a HtDP-based curricu-

5.1 Developing a Breadth-First Solution

The heuristic-based depth-first N-puzzle solver assumed a solution can be found
by always exploring the best successor of the current board. This assumption is
removed and all possible sequences starting at a given board are explored. This
requires that a list of all sequences generated so far be maintained. It is important
during the exploration of this idea in the classroom to have students realize
that this list of sequences must be maintained in order by length. Otherwise,
the strategy may degenerate into a depth-first search that leads to an infinite
recursion.

The implementation builds on the work done for the heuristic-based depth-
first N-puzzle solver. The function find-solution-bfs takes as input a board, \( b \),
and returns a sequence from \( b \) to \( \text{WIN} \). To accomplish this, a helper function,
search-paths, is called that takes as input an accumulator that stores the list of
all sequences generated so far. Initially, this list of sequences contains a single
list that contains \( b \). The function search-paths is a combination of generative
recursion and accumulative recursion. Each time the function is called, it checks
if the first board in the first sequence is \( \text{WIN} \) and, if so, it returns the first
sequence. Otherwise, the successors of the first board in the first sequence are
generated and new sequence is generated for each successor by adding it to the
Fig. 8. An implementation for creating an intial N-puzzle board.

front of the first sequence. To maintain the accumulator invariant, the list of sequences that does not include the first sequence is appended with the new sequences generated for the recursive call. A sample implementation is displayed in Figure 9.

Students must develop an accumulator invariant as well as an argument for termination. The accumulator invariant is developed, as mentioned above, during the exploration of the idea to search all possible sequences. The argument for termination hinges on having students realize that as paths get longer the number of moves required for one or more paths to reach the winning board gets smaller. Thus, the number of moves required to reach the winning board will eventually reach 0 for some path and the algorithm returns the appropriate sequence.

5.2 Refining the Solution: Deriving an A*-like Algorithm

The breadth-first N-puzzle solver does find a solution for any given board, but students soon discover that the help button is very sluggish and in some cases extremely so. The problem, of course, is that exploring all possible sequences starting at a given board is a great deal of work. Students can be lead to realize that after 10 moves the number of sequences being searched surpasses $2^{10}$ and after 20 moves surpasses $2^{20}$. This provides an opportunity to expose students to the problems of exponential growth. At this point, students are asked if searching all possible sequences and searching all possible sequences at the same time is necessary. This is a difficult question for them to answer. Most students will say...
yes to both questions, because all possible sequences must be searched. In other words, most students at this level are unlikely to realize on their own that not all sequences need not be searched nor that all sequences that ought to be searched have to be simultaneously searched.

There are two main ideas that are planted in students’ minds. The first idea is that not every sequence needs to be explored. We draw on the experience obtained from the depth-first N-puzzle solver. If any successor, \( s \), of a given board, \( b \), has been explored (i.e., the successors of \( s \) have been generated), then the path through \( b \) to \( s \) need not be explored. The reason is that a sequence, of equal or shorter length, to \( s \) has already been generated. The second idea is that we can choose to explore the most “promising” sequence first instead of blindly exploring all possible sequences at the same time. This leads the class discussion back to the Manhattan distance heuristic as a mechanism for deciding which sequence is the most promising. The idea to always explore the most promising sequence first is one that students in CS1 can grasp and implement.

Figure 10 displays an implementation of this strategy\(^3\). The function `search-paths` requires two accumulators each with its own invariant. The accumulator `visited` is a list of all the boards whose successors/children have been generated. The accumulator `paths` is a list of all the sequences starting at \( b \) that may need to be explored and that have no repeated boards in them. Both invariants, with some guidance, can be developed by students. The development of these

\(^3\) Due to figure size limitations, all comments including contracts, purpose statements, and accumulator invariants have been omitted.
(define (find-solution-a-star b)
  (local
    [(define (find-best-seq seqs)
        (cond [[(empty? (rest seqs)) (first seqs)]
          [else
            (local [[(define best-of-rest (find-best-seq (rest seqs)))]
              (cond [(< (manhattan-dist (first (first seqs))
                  (manhattan-dist (first best-of-rest)))
                (first seqs)]
                [else best-of-rest]]))]))
    (define (search-paths visited paths)
      (local [[(define bstseq (find-best-seq paths))]
        (cond [[(equal? (first best-path) WIN) bstseq]
          [else
            (local
              [[(define children
                  (filter (lambda (c) (not (member c visited)))
                (generate-children (first bstseq)))
              (define new-seqs (map (lambda (c) (cons c bstseq))
                children))]
              (search-paths
                (cons (first bstseq) visited)
                (append new-seqs (rem-path bstseq paths)))))]])
      (reverse (search-paths '() (list (list b))))))

Fig. 10. An A* N-puzzle solver.

invariants is likely to be the most time-consuming exercise in class. The rest of
the implementation flows faster. The code finds the best sequence in paths. If the
winning board has been reached by the best sequence, then the best sequence
is returned. Otherwise, the program filters the successors of the last board\(^4\) in
the most promising sequence to remove boards that have already been explored.
New sequences are generated using \texttt{map} to add each remaining successor to the
most promising sequence. Notice that both of these computations are achieved by
reinforcing lessons on abstraction that students have been exposed to in the near
past. Finally, to maintain the two accumulator invariants, the last board of the
most promising sequence is added to \texttt{visited} and the new sequences are appended
with sequences obtained from removing the most promising sequence from \texttt{paths}.
The only remaining tasks students must implement is finding the most promising
sequence and removing a sequence from a list of sequences. The first can be done
either by using accumulative recursion with an accumulator that remembers
the best sequence so far or using structural recursion. The implementation in
Figure 10 displays the latter and redesigning such a function using accumulative

\(^{4}\) Note that sequences are reversed making the last board in the sequence the first in
the list.
recursion is left as an exercise to give students more practice. The second is a straight-forward exercise using structural recursion. The algorithm developed is in essence an A*-like algorithm. That is, it is a combination of a breadth-first strategy and a depth-first with backtracking strategy. Such algorithms are commonly referred to as informed heuristic search strategies. What is most noteworthy is the fact that the development flows naturally from following the steps of the design recipe and iterative refinement. Students reason the algorithm into existence instead of being told about an algorithm. Such a development challenges the tacit assumption that A*-like algorithms are too complex for beginning students to understand and, therefore, are left as material restricted to more advanced courses such as an Introduction to Artificial Intelligence. There is, of course, one important observation about the N-puzzle domain that allowed us to simplify the design. Once a board is encountered there is no need to change its predecessor, because the cost of reaching it through the sequence of a previous encounter is always as good or better than the cost through the new sequence. In a full-fledged A* algorithm, the costs of the different sequences to a board must be examined to always maintain the sequence with the least cost.

6 Facilitating Deployment in the Classroom

The most important computational components of the presented N-puzzle solver have been developed in this article. The remaining components have to deal with the development of the interface with a player. The developers of HtDP have implemented a library (or teachpack as referred to by HtDPers), called universe, that allows students to easily develop interactive programs such as a video game. Universe envisions an animation as a series of snapshots of an evolving world. There is a clock that at every tick displays the next snapshot of the world. Students must define the elements of the world and define functions for computing the next snapshot of the world when the clock ticks or when an external event, such as a keystroke or a mouse movement, occurs. Students must also define functions for drawing the world and for detecting the end of the game/animation.

It is important to carefully gauge the amount of work that is asked of from beginning students. Although the universe library truly simplifies the development of video games, sometimes students feel overburdened by the fine details of deciding on what tile a mouse click has occurred or of drawing the N-puzzle with a help button. If such is the case, invariably students get bogged down by writing drawing and mouse processing functions which leads them to relegate to the back burner the important lessons about generative and accumulative recursion. After all, in the mind of a beginning student nothing makes sense if you can not play the game. When faced with such a problem, the best course of action is to eliminate the need for students to develop these low-level functions.

5 This function does not appear in Figure 10 due to space limitations for figures.
This can be achieved by writing a library/teachpack specifically for the N-puzzle problem. The teachpack ought to include all the functions necessary for drawing the puzzle with the help button and for processing mouse events as well as the interface with the universe teachpack. In this manner, students can focus on the important lessons of generative and accumulative recursion.

7 Related Work

The most closely related work on teaching generative and accumulative recursion to beginners is presented in HtDP. HtDP presents generative recursion as programs that have recursive calls that do not operate on part of the input. Instead, they generate a new instance of the problem. The examples used include, among others, moving a ball across a canvas, quick-sort, fractals, and the computation of the greatest common divisor (gcd) of two numbers. Of these, the only example that truly captures the imagination of students is fractals. The reason is that fractals allow for a student to personalize their solutions to problems. Problems like quick-sort and gcd, although important to be exposed to, do not permit for the personality of the student to be incorporated into their programs. Fractals and the N-puzzle video game, allow students to personalize solutions to their liking and that seems to be a great motivator by giving students a creative outlet to distinguish themselves and their work. The important lesson is to strike a balance between problems that allow personalization and those that do not. Both need to be included in a CS1 course. Problems that do not allow personalization, force students to focus on the lessons of designing functions that use generative recursion. Once those lessons have been presented and practiced, it is important to give students a chance to have a little fun with problems that allow personalization like the N-puzzle problem. In the N-puzzle problem, students can personalize the board (e.g., letters, number, images, etc.), the color of the tiles, and the definition of the winning board.

HtDP introduces accumulative recursion as a solution to the loss of knowledge between recursive calls. This can lead to efficiency issues in the case of programs designed using structural recursion or to problems not being solved in the case of generative recursion. The examples developed include finding a path in a graph and reversing a list. HtDP also outlines exercise that, like the work presented in this article, require students to combine skills to design programs that requires functions that exploit structural, generative, and accumulative recursion. None of the problems are video-game-based, but, in fairness, HtDP was published before the development of the universe teachpack.

To the best knowledge of the author, there have been no published attempts to have beginning students work on the N-puzzle problem nor on developing A*-like algorithms. The N-puzzle game has been used to motivate topics in Artificial Intelligence and Machine Learning [4]. In addition to using the N-puzzle in an undergraduate AI course, the authors report using the N-puzzle game in a data structures and an algorithms course. In contrast, the approach presented in this article targets beginning students.
The teaching philosophy put forth by HtDP applied to the design of functional video games is a powerful combination that allows CS1 students to receive a solid introduction to programming while at the same time to become enthusiastic about the field of Computer Science. The enthusiasm comes from seeing in practice that what they are learning in the classroom is directly applicable to a domain that is of interest to them. In addition, the video game domain allows students to personalize solutions which means that students are not all producing the exact same solution to problems. Contrast this to solving problems in a Mathematics, Physics, or Chemistry course and it is easy to see why students find working with video games fun, personally rewarding, and enlightening. There are examples in the literature that illustrate how to design animations and video games that require the use of primitive data, structures, and structural recursion. The work described in this article is an example of how, in the CS1 classroom, to make the transition from structural recursion to generative and accumulative recursion using a video game as motivation to capture student enthusiasm. The choice of game, the N-Puzzle, was made to also demonstrate that informed heuristic search strategies, traditionally the domain of undergraduate courses in Artificial Intelligence, are within the grasp of CS1 students. Students do not simply study such an algorithm. Instead, the full power of program by design allows CS1 students to reason such an algorithm into existence. If this work is replicated elsewhere, there is no doubt that it will be an exciting time for Computer Science education and it will elevate the relevance of functional programming in the minds of future CS professionals.

Future work includes demonstrating how functional video games can be an effective pedagogical tool for motivating and teaching parallel programming to CS1 students. Functional programming has been identified as providing a clear and concise way to program parallel computers and distributed computations [6,9]. It is time for this knowledge to reach down to the CS1 classroom. The approach will assume that students have a foundation using different forms of recursion as well as abstraction and will use the universe teachpack as in the work described in this article.

References


GiN: a graphical language and tool for defining iTask workflows

Research paper

Jeroen Henrix, Rinus Plasmeijer, and Peter Achten

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
j.henrix@science.ru.nl, {rinus, p.achten}@cs.ru.nl

Abstract. Workflow Management Systems (WFMSs) are software applications which coordinate business processes. The coordination is based on a workflow model, expressed in a domain-specific Workflow Description Language (WDL). WDLs are typically graphical languages because the specification has to be understandable for domain experts as well as workflow application developers. Commonly, only simple workflows can be described while additional coding is needed to turn the description into a running application. The iTask system is a combinator library, embedded in Clean, to construct WFMSs. Complex workflows can be defined declaratively from which a complete web-based application is generated. However, the textual specification is less suitable for domain experts who are used to graphical notations. In this paper we address this problem and present GiN: a graphical notation for iTask workflows, as well as a prototype implementation of a tool to construct GiN workflows interactively and graphically. The tool is fully integrated in the iTask system: it is just another iTask component, and workflows created with GiN can be subsequently added and executed dynamically as part of other workflows.

1 Introduction

In this paper we present GiN (Graphical iTask Notation). GiN is both a graphical notation for the iTask system as well as a tool to construct iTask workflows in an interactive and graphical way. The iTask system [18] is a combinator library, embedded in the pure and lazy functional programming language Clean, to construct Workflow Management Systems (WFMSs) in a functional style. WFMSs are software applications that coordinate business processes. This coordination is based on a workflow model: a formal description in a Workflow Description Language (WDL) of the tasks that comprise a business process.

Conventional WDLs have a graphical nature. This has as advantage that their notation is perceived as intuitive and can be used in the development process by both workflow engineers and domain experts. Frequently, these WDLs were based on (coloured) Petri nets [2].
In contrast, the iTask EDSL uses a textual WDL; workflow models are created using the CleanIDE. To understand and appreciate an iTask workflow model, one needs to be trained in functional programming. It is our goal to make the iTask system and formalism accessible for the workflow community. GiN is the first step in this project to achieve this goal.

The GiN language is a hybrid language that combines graphical elements with textual elements. Where suitable, the graphical elements are borrowed from graphical WDLs used in the workflow community. GiN adds a few new graphical elements that are particular to iTask. Right from the beginning we have decided that GiN is not a visual programming language substitute for Clean. That would defeat the purpose of GiN of being an accessible tool for domain experts who are not trained in functional programming. Furthermore, in contrast with conventional systems, the iTask system can generate a complete web-based application from a workflow specification. In conventional systems, the graphical definition of a workflow is only a partial specification that defines the control flow between tasks, and hence, a significant programming effort is required to implement the data dependencies and data flow. For these reasons we think it is reasonable that in GiN not all parts of a workflow can be expressed graphically. The current version of GiN is restricted to the ‘classic’ iTask combinator language core [18]. Recent extensions of the iTask API, such as run-time changes [19] and support for GUIs and shared data [15] are not supported yet. This belongs to future work.

The GiN tool is an interactive and graphical editor to create iTask workflows. It supports the user with direct feedback about mistakes in the workflow under construction. The GiN tool is integrated in the iTask system, and can be used like any other task in a workflow. We envision that this integration is particularly useful in the presence of change [19] when workflow engineers or managers need to design alternative workflows quickly and correctly.

The contributions of our work are:

- We define and motivate a new hybrid WDL, GiN, that combines the salient features of iTask with elements from the workflow community.
- We design and implement a prototype GiN tool that allows users to construct GiN workflows while being continuously informed about the correctness of the workflows under construction.
- We integrate the GiN tool in the iTask system. Workflows can invoke the tool, and use the output subsequently.

The remainder of this paper is organized as follows. First, we define the GiN language in Section 2. The GiN tool and its design choices are described in Section 3. Section 4 explains how to compile GiN specifications to executable workflow applications. Related work is discussed in Section 5. Section 6 presents conclusions and future work.
2 The GiN WDL

In this section we introduce the GiN language. We first give a concise introduction to the iTask WDL (Section 2.1), followed by the current design of the GiN WDL (Section 2.2), and an example of a higher-order workflow specified in GiN (Section 2.3).

2.1 The iTask WDL

The iTask WDL is a combinator language, constructing basic and composite tasks of abstract type Task a. Figure 1 shows an excerpt of the signatures of basic tasks and task combinators. The iTask WDL is generic: the generic type class constraint iTask allows the framework to automatically derive a fully operational web-based GUI for any value of any first-order data type. In Clean both overloaded and generic constraints are placed at the end of a type signature. For instance, in Figure 1, descr is a type constructor class, and iTask is a generic class. The semantics of iTask is formally defined [11, 19].

updateInformation :: d a → Task a | descr d & iTask a
showMessage :: d → Task Void | descr d
return :: a → Task a | iTask a

(updateInforma) infixl 1 :: (Task a) (a → Task b) → Task b | iTask a & iTask b
(showMessage) 1 :: (Task a) (Task b) → Task b | iTask a & iTask b

任何 Task :: Task a & Task b
anyTasks :: [Task a] → Task [a] | iTask a

(any :.) infix 3 :: User (Task a) → Task a | iTask a

Fig. 1. Excerpt of basic tasks and task combinators of the iTask WDL

The basic task (updateInformation descr init) generates a GUI to allow the user to update an initial value init; the task (showMessage descr) shows a message descr to the user and returns the Void value when terminated. The iTask WDL is monadic, and provides the usual monadic core combinators return, >>=, and >=>. Tasks can be composed in parallel. Either the result of the first completed task is returned (|&| and anyTask combinators) or the results of the parallel tasks are collected and returned as a whole (|&| and allTasks). Finally, a task t can be assigned to a user u with u @: t.

2.2 The GiN WDL

Although the iTask library is a textual formalism, a graphical notation comes naturally. In order to make GiN appealing for workflow engineers, we have
adapted graphical notation from a number of graphical WDLs of workflow sys-
tems. These are workflow nets [1], YAWL [3], event-driven process chains [9],
and UML activity diagrams [5]. We refer to [8] for a detailed discussion. Figure 2
enumerates frequently occurring elements that are found in these languages.
Workflow units are depicted as boxes (a-b), the control flow is depicted with
arrows (c), and parallel compositions are given structure with parallel split and
merge patterns (d-e).

![Workflow Diagram](image)

**Fig. 2.** Frequently occurring elements in graphical WDLs

In a GiN project, a workflow is defined by means of a collection of **task definition diagrams** and **host language modules**. A **Task definition** diagram (see picture to the right) introduces a task function of name \( f \) that uses parameters \( a_i \) of types \( \alpha_i \) \((i \geq 0)\). The names of task definitions must not overlap. The parameter-names \( a_i \) are simple variable names. The body of the task definition diagram is a digraph \( e \) with exactly one source node \((\text{source}(e)\) with in-
degree zero, indicated by \( \bullet \), and one sink node \((\text{sink}(e)\) of outdegree zero, indicated by \( \blacksquare \)). The digraph \( e \) consists
of nodes and edges, which are arranged according to the constructs specified in Figure 3.

We define the semantics of a GiN graph by means of a map \([\cdot]\), which is also given in Figure 3. Only GiN graphs that map to
well-formed and well-typed Clean expressions are well-formed themselves. Of
each GiN construct \( n \), we define the scope rules by means of two parameter-
ized sets: \( \text{in}(n) \) defines all known symbols immediately before entering \( n \)
and \( \text{out}(n) \) defines all known symbols immediately after exiting \( n \). If \( g \) is a
GiN graph, then \( \text{in}(g) = \text{in}(\text{source}(g)) \) and \( \text{out}(g) = \text{out}(\text{sink}(g)) \). Suppose
we have a **Task definition** diagram in an environment that defines identi-
fiers \( \text{env} \). If this diagram has name \( f \), arguments \( a_i \) \is\( \alpha_i \), and body \( g \), then
\( \text{in}(g) = \text{env} \cup \{ f \} \cup \{ a_i^{\alpha_i} \} \).

With a **Task application** construct \( n \), a task function \( f \) is applied to all
of its arguments. If \( f \) is defined by means of a **Task definition** diagram, then
the names of its parameters are repeated to guide the user to fill in the correct

\[
\begin{array}{c}
\text{f yields } \alpha \\
\text{a_1 is a } \alpha_1 \\
\vdots \\
\text{a_n is a } \alpha_n \\
\end{array}
\]

**TASK DEFINITION**
In a Task definition with name \( f \) and environment \( env \):

**Task Application**

\[
\begin{align*}
\text{let} & \quad a_1 = [e_1] \\
\text{...} & \\
\text{let} & \quad a_n = [e_n] \\
\text{in} & \quad [e]
\end{align*}
\]

\[= f \ [e_1] \ldots [e_n]\]

**Case**

\[
\begin{align*}
&\text{for } e_i, o :: \text{Task } \alpha : \\
&\quad \text{case } e \text{ of} \\
&\quad \quad p_i = [e_i] \\
&\quad \quad \_ = [o]
\end{align*}
\]

**Repeat Case**

\[
\begin{align*}
&\text{let } g \overset{a}{\rightarrow} = \\
&\text{case } e \text{ of} \\
&\quad p_i = [e_i] \text{ if } e_i \rightarrow t \\
&\quad \_ = [o] \text{ if } o \rightarrow t \\
&\quad \text{in } g \overset{a}{\rightarrow}
\end{align*}
\]

(with \( a \overset{a}{\rightarrow} \equiv \text{in}(s) \setminus \{f\} \setminus env \),
and \( g \) a fresh name)

**List Comprehension**

\[
\begin{align*}
&\text{foreach } x \text{ in } e \\
&\quad \text{given } p \\
&\quad \text{as } x - l \\
&\quad \text{as } x - t
\end{align*}
\]

\[= [[e] \setminus x - l] \setminus p\]

**Literal**

\[= e \text{ if } e \text{ is a Clean expression}\]

**Fig. 3.** GiN’s graphical language constructs and their mapping to textual iTask notation.

---

287

5
arguments. The names are absent in any other case, for instance when invoking an iTask API function such as updateInformation or showMessage. Each argument $e_i$ has access to the same set of symbols: $\text{IN}(e_i) = \text{IN}(n)$. The scope of the task application is closed, hence $\text{OUT}(n) = \text{IN}(n)$.

GiN supports two constructs for case distinctions: Case and Repeat case. The Case construct is actually a direct visualization of the case-of in the host language: given a literal expression $e$, perform case distinction by means of pattern-alternative pairs $p_i - e_i$ and an optional default case $o$ in the usual way. The pattern-alternative pairs are ordered from left to right, top to bottom, followed by the default case. The scope rules are: $\text{IN}(e) = \text{IN}(n)$, and for each pattern $p_i$: $\text{IN}(p_i) = \text{OUT}(e)$ and alternative $e_i$: $\text{IN}(e_i) = \text{OUT}(e) \cup \text{free}(p_i)$; the optional default case has $\text{IN}(o) = \text{IN}(n)$. The scope of the entire Case is closed, hence $\text{OUT}(n) = \text{IN}(n)$. The Repeat case construct $n$ extends the Case construct with the possibility for each alternative to connect back to an earlier node in the GiN graph. At least one alternative must be connected to the sink of $n$ to obtain a well-formed graph. In this way, structured loops can be created in GiN. The meaning of the Repeat case is given by means of a recursive function with name $g$ (which must be unique with respect to all symbols within the task definition diagram). The scope of the Repeat case is closed, hence $\text{OUT}(n) = \text{IN}(n)$. Locally, the following equations hold: $\text{IN}(e) = \text{IN}(n)$, for each $p_i$: $\text{IN}(p_i) = \text{OUT}(e)$, for each $e_i$: $\text{IN}(e_i) = \text{IN}(p_i) \cup \text{free}(p_i)$.

GiN adopts the host language construct let-in by means of the Let construct $n$. This construct introduces (or shadows) identifier-expression pairs $a_i = e_i$. This is useful in combination with Repeat case. The expressions $e_i$ are allowed to be mutually recursive. The body $e$ and each $a_i$ use the same identifiers: $\text{IN}(e) = \text{IN}(e_i) = \text{IN}(n) \cup \bigcup \{a_i\}$. Finally, $\text{OUT}(n) = \text{OUT}(e)$.

The List (comprehension) constructs provide a means to work with multiple tasks. A List $n$ supports direct enumeration of tasks $e_i$. Of each task element, $\text{IN}(e_i) = \text{IN}(n)$. A List comprehension $n$ uses a literal generator-list $l$ and literal filter predicate $p$ to enumerate tasks. $x$ is currently only allowed to be a variable. The free identifiers of $l$ and $p$ must be in scope: $\text{free}(l) \subseteq \text{IN}(n)$ and $\text{free}(p) \subseteq \text{IN}(n) \cup \{x\}$. The body $e$ can use the known names that are in scope, $x$ and the free names of $p$: $\text{IN}(e) = \text{IN}(n) \cup \{x\} \cup \text{free}(p)$. The scope of both list constructs is closed, hence $\text{OUT}(n) = \text{IN}(n)$.

All graphical WDLs denote the sequential order of tasks $a$ and $b$ as $a \rightarrow b$. The major concern of most graphical WDLs is control flow. The iTask WDL is different because it firmly integrates control flow and data flow. The key combinator is of course monadic bind, which is captured with the Bind $n$. The workflow identifies the data that is returned by task $e_1$ by means of a pattern $p$. The scope rules are: $\text{IN}(e_1) = \text{IN}(n)$, $\text{IN}(p) = \text{OUT}(e_1)$, $\text{IN}(e_2) = \text{OUT}(e_1) \cup \text{free}(p)$, and $\text{OUT}(n) = \text{OUT}(e_2)$. The Sequence construct $n$ is just the simplified version that omits the pattern. Its scope rules are: $\text{IN}(e_1) = \text{IN}(n)$, $\text{IN}(e_2) = \text{OUT}(e_1)$, and $\text{OUT}(n) = \text{OUT}(e_2)$. The Return $n$ captures the monadic return combinator. To support higher-order workflows, its value can be another GiN graph or a literal host expression. It’s scope is closed, hence $\text{OUT}(n) = \text{IN}(n)$. 

288
All graphical WDLs support several kinds of parallel composition of tasks. In GiN, this is dealt with by means of the PARALLEL (LIST) constructs. The PARALLEL is useful when dealing with a statically known (small) number of tasks, or when the tasks return values of different type. The PARALLEL LIST is useful when a dynamically known number of tasks need to be created, or when it is unfeasible to enumerate the individual tasks. Most graphical WDLs have the convention to use a split entity to denote the creation of parallel tasks, and a merge entity to merge and synchronize parallel tasks. Graphical WDLs that are based on Petri-Nets allow arbitrary split-merge patterns. GiN adopts the graphical convention of using split-merge patterns, but only allows block-structured patterns. For both kinds of constructs $n$, the scope is closed, hence $\text{OUT}(n) = \text{IN}(n)$. Within a PARALLEL $n$, each $e_i: \text{IN}(e_i) = \text{IN}(n)$. Within a PARALLEL LIST $n$, $\text{IN}(l) = \text{IN}(n)$.

A task $e$ is assigned to a user $u$ with the ASSIGN $n$. Here, $u$ is a literal host language expression. We have $\text{IN}(u) = \text{IN}(e) = \text{IN}(n)$ and $\text{OUT}(n) = \text{OUT}(e)$.

Finally, the LITERAL construct $n$ can be used to invoke any correct host language expression.

**Evaluation** The GiN language has been designed to make the iTask WDL more accessible for workflow engineers and domain experts. The question needs to be answered whether we have moved sufficiently close towards the workflow community, and, dually, whether we have not moved too far away from the iTask WDL. We have yet to conduct experiments to verify the first question, but we can answer the second question affirmatively. The GiN WDL is arguably simpler than the full-blown textual iTask WDL: we do not allow currying in task applications, pattern-matching and guards are absent in task definitions, and list comprehensions are limited to a single generator-filter. However, the GiN WDL still preserves the salient iTask features: parameterized tasks and higher-order tasks can be defined and used, recursive workflows can be defined, and control flow and data flow are tightly integrated.

### 2.3 Example

To illustrate GiN, we implement an *English auction* workflow. In this type of auction, a number of bidders compete to purchase a ware. They are requested to place bids of increasing price. This process is controlled by an auctioneer who starts the opening bid. The process is terminated by the auctioneer when no bidder puts forward a higher bid. First, we define a Bid record type:

```
:: Bid = { user :: User // bid is done by bidder or terminated by auctioneer
          , ware :: String  // name of merchandise (≠ “”)
          , price :: Int // price (> 0)
```

The auction workflow is defined in a TASK DEFINITION (Figure 4). The auction workflow yields a bid. The first parameter of auction, bidf, abstracts over the concrete way of bidding (an example could be updateInformation "Make your bid")
and illustrates the use of higher-order tasks. The auctioneer parameter is a User who is in charge of the auction process, and starts with the opening bid. The bidders parameter is a list of users [User] who compete to purchase a ware. Finally, current is the currently pending Bid.

The auction workflow is an iterative process (REPEAT CASE) in which bidders and auctioneer interact. With a LIST COMPREHENSION the participants (the auctioneer and all bidders, denoted as [auctioneer:bidders]) are collected. Each participant is assigned the same task (using ASSIGN): to enter a new bid according to the higher-order task function bidf (using TASK APPLICATION). The PARALLEL LIST uses the ∨₁ˢᵗ merge node to determine the first person who enters a new bid. The value of the bid is identified with the variable newBid (using BIND). The topmost case distinction needs to inspect the new value newBid.price. If the price is higher, it becomes the current bid (using LET and looping back to the start of the auction graph). If not, the person newBid.user who placed the bid is checked. If it was not the auctioneer, the bid is invalid, and the process loops back to the start of the auction graph without changing any value. Otherwise, the workflow terminates and returns the most recent bid (using RETURN).

For completeness, we show the [auction] translation:

```
auction :: (Bid → Task Bid) User [User] Bid → Task Bid
auction bidf auctioneer bidders current =
    let g bidf auctioneer bidders current =
        anyTask [b @: bidf current] \ b ← [auctioneer : bidders] >>= λnewBid →
            case newBid.price > current.price of
                True = let current = newBid in
                    g bidf auctioneer bidders current
                False = case newBid.user = auctioneer of
                    True = return current
                    False = g bidf auctioneer bidders current
            in g bidf auctioneer bidders current
```
Note that the translation is different than a typical task function manually written by an iTask programmer. Because the outermost construct is a REPEAT CASE, the local function g is actually redundant. Besides, the Clean language allows us to express the conditions more compactly using guards, which are not available in GiN.

3 The GiN tool

In order to investigate whether the GiN language is useful in practice, we need a tool to create and compile GiN diagrams. In this section we discuss the editor component with which users can create and maintain GiN diagrams. The compiler component is discussed in section 4.

The GiN tool is a proof-of-concept implementation. It is based on the Oryx editor, which is part of the Oryx platform [4]: an academic open source framework for business process management. The Oryx editor offers the standard functionality we expect from a graphical editor. We use Oryx for a number of reasons. First, reimplementing this kind of standard functionality is a duplication of effort, and Oryx is already a stable and usable editor. Second, Oryx is easily extended with support for new workflow definition languages, by means of “stencil set” definitions. Third, Oryx is web-based and uses web technologies that are deployed in iTask, like the Ext JS web framework and the JSON (JavaScript Object Notation) data interchange format, which facilitates integration.

Figure 5 shows a screenshot of the GiN tool. The editor consists of a drawing canvas and a repository. The drawing canvas shows the Task definition under construction. The repository shows available basic and user-defined tasks, as well as separate split and merge connectors for the Parallel (List) and Repeat (Case) constructs. Workflows are compositional: both from textual and graphical workflow definitions, one can import other (textual or graphically defined) modules.

While the user is constructing a workflow, the editor continuously provides her with informative feedback. GiN diagrams can contain two sorts of errors: either the diagram does not conform to the structure defined in Figure 3, or (combinations of) literal Clean expressions are erroneous. The first sort of error is detected by the compilation process of a GiN diagram under construction to iTask code, and the second sort of error by invoking the Clean compiler on the generated code. Due to the speed of the Clean compiler, this process is not excessively time-consuming. On a test system (Intel Core 2 Duo, 2.1 GHz), we measure compilation times in the order of 100 milliseconds, which is acceptable for interactive use. The GiN tool parses error messages that may have been generated in the two steps and indicates the source of the error on the right spot in the GiN diagram under construction. An example indication is visible in Figure 5, where the diagram contains an undefined variable name.

The GiN tool is integrated in the iTask system. This has several consequences. First, from the point of view of the iTask system, the GiN tool ‘is just another editor’ for values of type GiNDiagram (the internal data structure that is used
to represent GiN diagrams). By adding \texttt{(updateInformation "Create your workflow!" myGiNDiagram)} in any workflow definition results in the creation of the GiN tool to allow the user to work on a GiN diagram with initial value \texttt{myGiNDiagram}. As a consequence, it can be used in a meta workflow: an iTask workflow which defines how new workflows have to be constructed. One can think of scenarios where some people construct workflows with the GiN tool, while others have to approve the workflow thus designed. Second, as part of such a scenario one also wants to use such a new workflow once it has been approved. This means that the resulting code corresponding to the new workflow has to be dynamically plugged into the running iTask application. Clean facilitates this by making use of dynamic types \cite{17} and the ability to link any value of a dynamic type into a running application \cite{23}. Third, this technology allows the user to browse her repository of created workflows, and use them to build new workflows, just by drag-and-drop in the user interface of the tool. Fourth, in the iTasks project, a first step towards adaptability has been set, by making the technical abilities to make changes to running tasks \cite{19}. We envision that the GiN tool is particularly useful to make changes right on the spot, in an understandable way for end users.

4 The GiN compiler

In this section we discuss the compilation process of GiN diagrams under construction to iTask workflows. The basic idea is that the GiN tool serves as a preprocessor for the Clean compiler (Figure 6). The preprocessor detects ill-structured GiN diagrams and generates error messages. Well-structured GiN diagrams are compiled to textual iTask source files. The Clean compiler detects
remaining errors and generates error messages. Correct iTask programs are compiled to an executable that generates a dynamic on disk that can be used for further processing, as described in section 3.

Internally, a GiN diagram is represented as a graph structure of type \texttt{GiNDiagram}, consisting of a set of nodes and a set of edges. In order to represent iTask source files, we defined an abstract syntax tree (AST) containing only the elements needed for the mapping. These elements are function definitions, literals, variables, prefix and infix applications, lambda abstractions, case expressions, let-expressions, tuples, lists and simple (one-generator) list comprehensions. Literal Clean expressions found in diagrams (\texttt{Literal}) are stored as literal text in the AST. By pretty-printing the AST, we obtain an iTask source file.

The preprocessor has to make a transformation from the nodes and edges in the \texttt{GinDiagram} graph, to expressions in the AST. Looking at Figure 3, we observe that some of the constructs consist of separate split and merge nodes, namely in \texttt{Case}, \texttt{Repeat Case} and \texttt{Parallel}. These nodes are are mapped pair-wise to an iTask expression. Given the fact that constructs may be arbitrarily combined, the question is how to identify the corresponding nodes that should map to an expression. A complicating factor is that the graph structure may contain cycles to express loops (as in \texttt{Repeat Case}).

A similar problem is more widely known in the workflow community. Many WDLs used in business modeling are \textit{graph based}: these WDLs use a graph structure which allows arbitrary unstructured connections between nodes. Execution-oriented WDLs are often \textit{block based}, allowing only structured compositions of well-nested blocks. When mapping a graph-based WDL onto a block-based WDL, the structured parts need to be identified in order to map them. A structured part has a unique entry node and a unique exit node.

In [22], Vanhatalo \textit{et al} describe an algorithm that decomposes a graph $G$ into a \textit{Process Structure Tree} (PST). The branches in the PST are the structured subgraphs, while the leaves are subgraphs containing individual nodes from the original graph. The decomposition from a graph to a PST can be computed in linear time. We use this algorithm to parse GiN graphs. We added two restrictions: the exit node of a subgraph cannot be a \texttt{case split} node (to ensure each case alternative has an expression), and individual \texttt{let}, \texttt{split} and \texttt{merge} nodes are not decomposed to subgraphs. As an example, we show the identified subgraphs of the auction workflow (labeled $A...D$), the individual nodes (labeled $e...l$), together with its corresponding PST in Figure 7.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{diagram.png}
\caption{Compilation steps}
\end{figure}
Each subgraph $G'$ in the PST decomposition, with source node $s$ and sink node $t$, is classified in one of four categories:

- **Trivial subgraphs**, which consist only of a single node, so $s = t$.
- **Structured parallel subgraphs**, in which $s$ is a parallel split node, $t$ is a parallel merge node, $s$ splits to nodes $a_i$, and $a_i$ merges to $t$ ($1 \leq i \leq n$).
- **Sequential subgraphs**, which do not contain any parallel split or parallel merge nodes. Hence, these subgraphs are purely sequential, but may contain non-block structured branches and arbitrary cycles.
- **Other subgraphs**, which do not fit any of the above categories.

In the auction example, $A$ is a sequential subgraph, $B$ a structured parallel subgraph, and $C$ and $D$ are trivial subgraphs. Subgraphs in the first three categories can be mapped to iTask expressions. Other subgraphs from the fourth category are rejected, and an error is reported to the user. Such graphs may contain parallel branches which are not properly block structured, and thus cannot be expressed in terms of the ‘classic’ set of core iTask combinators. These cases could possibly be expressed using recent extensions of the iTask API [15], but this belongs to future work.

The mapping $\text{graphToExpr}(G)$ from a subgraph to an iTask expression is defined as follows:

- **Trivial subgraphs** are mapped straightforwardly according to the $\llbracket \cdot \rrbracket$ mapping of Figure 3. Note that the actual parameters of a TASK APPLICATION, elements in a LIST and the output expression of a LIST COMPREHENSION are GiN diagrams themselves, which are mapped recursively.
- In **Structured parallel subgraphs**, the split node $s$ and merge node $t$ are mapped pair-wise to a corresponding iTask combinator, according to the mapping $\llbracket \cdot \rrbracket$ of Figure 3. Each of the parallel branches is mapped recursively. If no matching pair of $s$ and $t$ is found in $\llbracket \cdot \rrbracket$, an error is reported to the user.
- **Sequential subgraphs** may contain arbitrary cycles. The idea is to unfold the paths starting from merge nodes (●) into separate functions, defined in let-expressions. Each edge entering such a merge node is translated to a tail-recursive call of the corresponding function. We traverse the subgraph starting from the source node. If we encounter another merge node, we call...
the corresponding function defined in the let-expression and stop. If we encounter another node, we map it according to \( \cdot \) of Figure 3. The bodys of the functions in the let-expressions are obtained in an analogous way, by traversing the subgraph starting from the nodes succeeding each merge node. More formally: let \( G' \) be a sequential subgraph with source \( s \), sink \( t \) and merge nodes \( m_1 \ldots m_k \neq t \). We define the predicate \( \text{ends}(n) = n \) is a merge node and either \( n \) is a sink node, or there is a path from \( n \) to a sink node consisting only of merge nodes. For each merge node \( m_i \) in \( G' \) for which \( \text{ends}(m_i) \) does not hold, define a new let-expression \( f_i \) with arguments \( \{f_i\} \setminus \text{env} \) and body \( \text{seqNodeToExpr}(G, \text{successor}(m_i)) \). We define \( \text{seqNodeToExpr}(G, n) \) as a mapping from an individual node \( n \) in a sequential subgraph \( G \) to an expression, depending on the context in which \( n \) occurs. \( \text{seqNodeToExpr} \) is shown in Figure 8.

<table>
<thead>
<tr>
<th>pattern</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( \text{graphToExpr}(n) )</td>
</tr>
<tr>
<td>( n = t ) \n ( \text{if } t \neq \bullet )</td>
<td>( f_i ) \n ( \text{in} ) \n ( {f_i} \setminus \text{env} )</td>
</tr>
<tr>
<td>( n )</td>
<td>( \text{case } e ) \n ( \text{of} ) \n ( p_1 = \text{seqNodeToExpr}(G, e_1) ) \n ( \vdots ) \n ( p_k = \text{seqNodeToExpr}(G, e_k) ) \n ( o = \text{seqNodeToExpr}(G, o) )</td>
</tr>
<tr>
<td>( n \rightarrow n' ) \n ( \text{if } n = ) \n ( {a_1 = e_1 ) \n ( \vdots ) \n ( a_k = e_k ) \n ( \wedge \neg \text{ends}(n') )</td>
<td>( \text{let} ) \n ( a_1 = \text{graphToExpr}(e_1) ) \n ( \vdots ) \n ( a_k = \text{graphToExpr}(e_k) ) \n ( \text{in} ) \n ( \text{seqNodeToExpr}(G, n') )</td>
</tr>
<tr>
<td>( n \rightarrow n' ) \n ( \text{if } \neg \text{ends}(n') )</td>
<td>( \text{graphToExpr}(n) \gg \lambda p \rightarrow \text{seqNodeToExpr}(G, n') )</td>
</tr>
<tr>
<td>( n \rightarrow n' ) \n ( \text{if } \neg \text{ends}(n') )</td>
<td>( \text{graphToExpr}(n) \gg</td>
</tr>
<tr>
<td>( n \rightarrow n' ) \n ( \text{if } \text{ends}(n') )</td>
<td>( \text{graphToExpr}(n) )</td>
</tr>
</tbody>
</table>

Fig. 8. Mapping \( \text{seqNodeToExpr}(G, n) \) of nodes in sequential subgraphs to expressions

5 Related Work

Many graphical WDLs are based on directed graphs and primarily model control flow. Data flow is added as an additional layer, in which all data is often glob-
ally accessible and stored in databases. Specifications are often partial, and hence require a significant additional software engineering effort in order to create a complete executable WFMS. There exist numerous graphical WDLs: industry standards like Event-driven Process Chains (EPCs) [13], UML activity diagrams [5], BPMN [16], and proprietary WDLs from WFMS vendors. As explained in section 2, the GiN language adopts notational conventions found in these graphical WDLs. Because GiN is based on the underlying iTask WDL, GiN diagrams have an executable semantics and express both control flow and data flow. Workflow data is strongly typed. Recursive and higher-order workflows can be defined graphically. GiN diagrams are hybrid, and can embed Clean code to express more complex business rules.

BPEL is a text-only, XML-based industry standard for expressing executable workflows based on web services. Several vendors invented their own (different) graphical tools for BPEL, like ActiveBPEL designer and Eclipse BPEL designer. Another approach is mapping BPEL to an existing graphical notation, like EPCs [14] or a subset of BPMN [21]. These systems focus on the coordination of web services. In the GiN/iTask system, web services can be included as basic tasks as well. The expressive power of the BPEL coordination language is, compared to GiN/iTask, relatively simple. Specification of user interaction and form handling is not part of BPEL.

We may consider the GiN language to be a visualization of a subset of a functional language, with the intent to make the language more accessible for users with limited programming knowledge. In this respect, it is similar to projects like Visual Haskell [20], VFPE [10], Vital [7], Eros [6], and Sifflet [24]. However, these visualizations are designed to be complete visual functional programming languages. Therefore their granularity of diagram language is much more fine grained than that of GiN, which is designed specifically to support the iTask WDL in a graphical way.

The way the GiN tool provides immediate feedback to inform the user about errors in the GiN diagrams is very much related to the continuous validation approach by Kühne et al [12]. They define a formalism in which a set of validation rules can be specified that are checked against an EPC model under construction. In our approach syntactic mistakes are detected in the preprocessor, and semantic mistakes by the Clean compiler.

6 Conclusions and future work

The GiN language is a new WDL that mixes graphical elements (inspired by the workflow community WDLs) with textual elements (inspired by the functional host language) and that connects control flow with data flow. With the GiN tool users can create GiN workflows. During this process, they are continuously given feedback about the correctness of the diagrams under construction. The next step in this project is to verify whether the GiN language and tool is indeed appreciated by domain experts when it is used in concrete projects.
GiN workflows are complete: when the specification is approved by the compiler, a runnable workflow is generated from it. Such newly defined approved workflows can be dynamically added to a running WFMS. The GiN tool is ‘just another iTask editor’. This allows one to define meta workflows: workflows for defining workflows. Consequently, the construction, approval and use of new workflows can be formally defined for a particular organization.

References


18. Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifi-
cations of interactive work flow systems for the web. In Ralf Hinze and Norman Ramsey, editors, Proceedings of the International Conference on Functional Pro-


Effective Test Set Generation

Pieter Koopman and Rinus Plasmeijer

Institute for Computing and Information Sciences (ICIS), Radboud University Nijmegen, the Netherlands
{pieter, rinus}@cs.ru.nl

Draft

Abstract. This paper under construction reports experience with model-based testing using our automatic test system \texttt{Gyst}. Based on a specification \texttt{Gyst} generates test cases, executes the associated tests, and gives a verdict. A unique feature of \texttt{Gyst} is that it is able to use specifications based on logical properties as well as state machines. Effective testing depends on smart specifications and effective generation of test suites by \texttt{Gyst}. Although the test library provides an generic algorithm that is able to derive generation for every first order type, the generation of test suited can often be optimized by simple means. In this paper we list some of the lessons learned over the years.

1 Introduction

In functional programming there are tools for property based testing available for over a decade. These tools generate test cases based on a logic expression as specification. The test tools try to falsify the given property by finding counterexamples. The generated test cases are used as values for the universal quantified variables in the logic expression. The property is a function that yields a Boolean, or some equivalent type. The arguments of the function are interpreted as the universal quantified variables. The test system evaluates the property for the generated test values and checks the result. If the function yields false for one of the generated test cases the property is falsified and the test system reports this. When all executed tests yields true the property is not falsified and the test system usually reports a pass.

Examples of these tools are QuickCheck [3], and SmallCheck [9] for Haskell, and Gast [5] for Clean. This property based approach to testing was ported to many other languages, see \url{http://en.wikipedia.org/wiki/QuickCheck} for an overview. The capabilities of the host languages often impose restrictions on the ported test system. However, the main approach remains unchanged: the user specifies a universal quantified logical property and the test system tries to falsify this property by automatically generating test cases. Although this approach to testing clearly set a trend, the idea of automatic generation of test cases and test scripts itself is much older. See e.g. [2,1,4].

When a property is falsified the test system has found an issue. Such an issue might indicate an error in the tested software. Also incorrect properties or
invalid test values might cause the test system to report issues. Whether or not a property that does not hold is indeed falsified depends critical on the test suite generated. Hence, effective test suite generation is critical for effective testing. The problem is of course that we do not know where the counterexamples are in the domain of the tested function, otherwise there is no need to test at all.

QuickCheck and most of its ports rely on pseudo random generation of test values. Using the primitives provided by the test system the test engineer has to specify how the actual test data are generated for all user defined data types used in the properties. This approach to test suite has two drawbacks. First, it requires effort and experience to generate effective test data. Generating appropriate test data is often a nontrivial task. The basic problem is that the set of possible test cases is often very large. For recursive types, that are heavily used in functional programming, the set of possible test values is even infinite. The pseudo random generation of test case often needs guidance to find counterexamples in a reasonable amount of time. Second, the counterexamples found are often not the minimal values that falsify the property. Larger counterexamples make it usually harder to find the source of this issue: either an error in the tested program, or a problem with the tested property. In order to reduce the problem of large counterexamples newer variants of the test system provides *shrinking*. Shrinking is an algorithm that produces a test set of smaller values based on the counterexample found. If the counterexample found is not the minimal counterexample, it is likely that shrinking can find a smaller counterexample for the same property. If the test suite contains lists, shrinking can for instance generate list based an a found counterexample by omitting systematically one or more elements from the given counterexample.

Another approach to test suite generation is based on the observation that if a property is falsifiable there is almost always a small counterexample. This is explained by the fact that functions over recursive types are usually fail because of an incorrect or missing alternative. There are usually small function arguments that selects the erroneous alternative and hence falsify the correctness property. Based in this observation the generic generation strategy of *Gyst* generates test suites with values ordered from small to large [6]. In this generic generation algorithm the size of a test value is determined by the number of constructors and basic values it contains. The test data generation of *SmallCheck* limits the nesting depth of test data in order to ensure that small test values are generated. Although it is easy to construct functions that have no small counterexamples, the generation of test values from small to large appears to be very effective in practise.

Systematic generation of test values instead of pseudo random generation also has the advantage that duplicated test values can be prevented. For small finite domains it is possible to detect that the test system has done an exhaustive test and hence the property is proven rather than passes the tests.

In this paper we report on our experience with test data generation over the last eight years is small examples as well as large projects. Based on this experience we give tips and tricks to generate effective test sets and describe
the heuristics implemented in G∀st to find counterexamples quicker. Since it is almost ever possible to construct artificial situations where these heuristics does not work, it is very hard to measure their effect objectively. Even when we would generate software to be tested and measure the number of tests needed to find a counterexample, there is a strong bias by the algorithm used to generate software to be tested. The final version of this paper will report on our going effort to measure their effect in the programs students wrote during a programming course. Maybe the typical flaws in these programs are quite different from the mistakes made by seasoned programmers.

2 Polymorphic Datatypes

Many functions and (abstract) data types written in functional programming languages are polymorphic. They are supposed to work correctly for any argument type, or for any argument type belonging to some type class. Even in Object Oriented languages like C++ and Java one is using more and more libraries with a type arguments. In C++ these are called templates and in Java it is called generic programming (which is quite confusing for the functional community).

During the tests we cannot leave this argument type unspecified. We have to use ordinary values as arguments for the tested functions, and hence choose some type to be used as argument for our polymorphic functions. It might be tempting to use something ‘simple’ like integers or Peano numbers as argument in unrestricted polymorphic situations. Since the polymorphic function to be tested has to work for any argument type it seems fair enough to spend no effort on this argument. Nevertheless it is worthwhile to be a bit more careful here. A function that has to work for any type argument cannot do much interesting things with these arguments. Usually using a type with only a few elements has the advantage that there are also a very small number of test cases with a specific size. However, we have to be careful to make the type used not so small that interesting behaviour or error detection becomes impossible. The total number of potential test cases for recursive types remains infinite. We illustrate this with two examples.

The first example is an ordinary FIFO queue. We assume that the queue has some smart amortized $O(1)$ implementation [7] that is worthwhile testing. The queue provides the following interface:

```plaintext
newQueue :: Queue a
enqueue :: a (Queue a) -> Queue a
dequeue :: (Queue a) -> (Maybe a, Queue a)
isEmptyQ :: (Queue a) -> Bool
head :: (Queue a) -> Maybe a
```

In order to test the implementation we state some general properties about FIFO queues:
A new queue is empty.

Enqueueing any element \( \mathit{b} \) in an empty queue and applying dequeue to the obtained queue should yield the value \( \mathit{Just \ b} \).

In a similar way \( \mathit{p1a} \) checks that the queue is empty after these actions.

In \( \mathit{p1b} \) we state that after queueing one element the queue is not empty.

We need to queue at least two elements to determine the difference between a stack and a queue. In this property we check whether the first element dequeued after queueing two elements is indeed the first element.

In the same spirit we can check that the queue remembers more than the first element. We enqueue two elements and dequeue two times, the obtained element should be the last element queued.

The final property in this example states that the function \( \mathit{head} \) applied to any queue should yield the same result as the first element of dequeue applied to that queue.

These properties can be directly converted to the Clean [8] equivalents needed by Gast. In order to be flexible in the type of elements queued we will use the type \( \mathit{T} \) for that type. As usual \( \mathit{G\forall\text{st}} \) treats function arguments as universal quantified variables. For instance \( \mathit{p1} \) states in mathematical terms \( \forall \mathit{b} \in \mathit{T} . . . \). . . .

\[\begin{align*}
p0 & : : \mathit{Bool} \\
p0 &= \text{isEmptyQ newQueue} \\
p1 & : : \mathit{T} \to \mathit{Bool} \\
p1 \mathit{b} &= \text{fst} (\text{dequeue} (\text{enqueue} \mathit{b} \text{ newQueue})) = \text{Just} \ \mathit{b} \\
p1a & : : \mathit{T} \to \mathit{Bool} \\
p1a \mathit{b} &= \text{not} (\text{isEmptyQ} (\text{enqueue} \mathit{b} \text{ newQueue})) \\
p1b & : : \mathit{T} \to \mathit{Bool} \\
p1b \mathit{b} &= \text{isEmptyQ} (\text{enqueue} \mathit{b} \text{ newQueue}) \\
p2 & : : \mathit{T} \mathit{T} \to \mathit{Bool} \\
p2 \mathit{b} \mathit{c} &= \text{fst} (\text{dequeue} (\text{enqueue} \mathit{c} (\text{enqueue} \mathit{b} \text{ newQueue}))) = \text{Just} \ \mathit{b} \\
p3 & : : \mathit{T} \mathit{T} \to \mathit{Bool} \\
p3 \mathit{b} \mathit{c} &= (\text{fst} \circ \text{dequeue} \circ \text{snd} \circ \text{dequeue} \circ \text{enqueue} \mathit{c} \circ \text{enqueue} \mathit{b}) \text{ newQueue} \\
&= \text{Just} \ \mathit{c} \\
p4 & : : (\text{Queue} \ \mathit{T}) \to \mathit{Bool} \\
p4 \mathit{q} &= \text{head} \ \mathit{q} = \text{fst} (\text{dequeue} \ \mathit{q})
\end{align*}\]

Note that these properties only use the functions in the interface. Without advanced timing we cannot tell if this implementation uses indeed some advanced \( O(1) \) implementation. The properties do check whether the type \( \mathit{Queue} \ \mathit{a} \) obeys some required properties for FIFO queues.

Before we can execute these tests we have to choose a concrete type for \( \mathit{T} \). Without that choice \( \mathit{G\forall\text{st}} \) cannot generate the desired arguments. Choosing a type like \( \mathit{Int} \) or \( \mathit{Peano} \) seems attractive, but would generate way too many test
cases for out properties. There is hardly any reason to check whether our Queue is able to distinguish all our $2^{32}$ or $2^{64}$ integers. In practice our test system will evaluate the test 'only' for the required number of tests (by default 1000). The other extreme is to use a one element type like $:: \mathbf{T} = \mathbf{T}$. With such a type it would be impossible to discriminate between a FIFO queue and a stack. In this situation a type with a few elements like $\mathbf{Bool}$ will work excellent. Gast can check whether the abstract type behaves indeed as a FIFO queue, but does not spend needless energy in enqueue and dequeue a large number of integers.

$:: \mathbf{T} := \mathbf{Bool}$

Using this type $\forall \mathbf{st}$ will prove all properties apart from the last one, p4, by exhaustive testing.

For property p4 $\forall \mathbf{st}$ needs to be able to generate queues as test argument. This can be done by the generic algorithm of $\forall \mathbf{st}$. The only problem is that we want to treat the queue as an abstract data type. The generic algorithm needs access to the actual type definition in order to apply the generic algorithm. The solution is to define the generation of instances in the implementation module by

derive ggen Queue

and to export this definition. Note that this does not break the abstraction barrier. Below we will discuss alternatives for this approach that does not require any changes in the abstract type tested.

With these properties we can test an implementation of the queue. A typical result is:

"p0" Proof: success for all arguments after 1 tests
"p1" Proof: success for all arguments after 2 tests
"p1a" Proof: success for all arguments after 2 tests
"p1b" Proof: success for all arguments after 2 tests
"p2" Proof: success for all arguments after 4 tests
"p3" Proof: success for all arguments after 4 tests
"p4" Passed after 1000 tests

For the interface queue we made some different implementations. The first one is correct, all others contain more or less sophisticated errors. We tested these properties for all implementations, in order to determine the error detecting quality of our properties. The results are listed in table 1.

It is obvious that other (abstract) data types can be tested in the same way. In general it is a good idea to use the smallest data type appropriate. This is not always the type Boolean. If we would have to test a polymorphic implementation of search trees without duplicates, Boolean is inappropriate, the test system would encounter at most four different trees during the tests and the maximum depth of the trees encountered will be two. For such a test the characters are a much better choice. It is small enough to allow proofs by exhaustive testing of some properties, yet big enough to allow many interesting trees.
3 State machines

Instead of writing properties that check individual interface functions from the queue, or simple combinations of these functions, we can also model the behaviour of the queue as a state machine. We introduce a tailor made data to encode the various manipulations of the queue. The elements in the type Qcmd, for Queue command, directly mirror the functions in the queue interface.

:: Qcmd a = EnQ a | DeQ | Head | IsEmpty

For generating inputs we rely on the generic mechanism of ∀st.

In order to specify the expected answers of the queue we need some simple model of the queue. We create such a model by implementing the queue by a list. This implies that enqueueing is order \( O(N) \) instead of \( O(1) \), as in a proper implementation, but for a model this does not harm us. The model specification is only used during testing to predict the required results. Obvious correctness of the specification is much more important than efficiency. With an inefficient specification testing takes longer, while an incorrect specification can miss errors in the software under test, SUT.

The result of the specification function represents tuple of the allowed list of outputs and the new state of the specification. The actual state of the SUT is hidden in the abstract type Queue and remains invisible during the tests. When there are very many outputs allowed it is convenient to model the allowed outputs by a function, the function after a Ft constructor captures this behaviour. Here it is easy to list all allowed outputs and states and we use the Pt label.

:: Spec state input output := state input → [Trans output state]
:: Trans output state = Pt [output] state | Ft ([output]→[state])

The functions of the queue yield two different types; the function isEmpty yields a Boolean, while all other functions yield a Maybe a and possible a new queue. It is often convenient to define a tailor made data type to capture all possible results. Here we use the data type :: EITHER a b = LEFT a | RIGHT b to handle the two options in one type. The specification of the queue machine reads:

spec :: [T] (Qcmd T) → [Trans (EITHER Bool (Maybe T)) [T]]
spec q (EnQ a) = [Pt [] (q++[a])]
spec [] DeQ = [Pt [RIGHT Nothing] []]
spec [a:x] DeQ = [Pt [RIGHT (Just a)] x]
spec [] Head = [Pt [RIGHT Nothing] []]
spec [a:x] Head = [Pt [RIGHT (Just a)] [a:x]]
spec q IsEmpty = [Pt [LEFT (isEmptyQ q)] q]
spec q _ = []

Pack the queue in a state machine to tests all transitions in a single specification.

qstm :: (Queue T) (Qcmd T) → ([EITHER Bool (Maybe T)], Queue T)
qstm q (EnQ a) = ([], enqueue a q)
qstm q Head = ([RIGHT (head q)], q)
qstm q IsEmpty = ([LEFT (isEmptyQ q)], q)
qstm q DeQ
Table 1. Testing Queue implementations with Booleans

<table>
<thead>
<tr>
<th>Test</th>
<th>Queue1</th>
<th>Queue2</th>
<th>Queue3</th>
<th>Queue4</th>
<th>Queue5</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>Proof</td>
<td>Proof</td>
<td>Proof</td>
<td>Proof</td>
<td>Proof</td>
</tr>
<tr>
<td>p1</td>
<td>Proof</td>
<td>Proof</td>
<td>Proof</td>
<td>Proof</td>
<td>Proof</td>
</tr>
<tr>
<td>p1a</td>
<td>Proof</td>
<td>Fail 1</td>
<td>Proof</td>
<td>Proof</td>
<td>Proof</td>
</tr>
<tr>
<td>p1b</td>
<td>Proof</td>
<td>Fail 1</td>
<td>Proof</td>
<td>Proof</td>
<td>Proof</td>
</tr>
<tr>
<td>p2</td>
<td>Proof</td>
<td>Proof</td>
<td>Fail 8</td>
<td>Fail 13</td>
<td>Fail 7</td>
</tr>
<tr>
<td>p3</td>
<td>Proof</td>
<td>Proof</td>
<td>Fail 2</td>
<td>Fail 17</td>
<td>Fail 6</td>
</tr>
<tr>
<td>p4</td>
<td>Passed</td>
<td>Passed</td>
<td>Fail 6</td>
<td>Fail 61</td>
<td>Fail 65</td>
</tr>
<tr>
<td>spec</td>
<td>Pass</td>
<td>Fail 2</td>
<td>Fail 17</td>
<td>Fail 6</td>
<td>Fail 6</td>
</tr>
</tbody>
</table>

♯ (mba, q2) = dequeue q
= ([RIGHT mba], q2)

G\vast test such a state machine by pseudo random choosing transitions allowed in the specification. The corresponding input is applied to the SUT. The transition relation requires that the observed output of the SUT is allowed by the specification.

Using Booleans as queue elements and a deliberately incorrect implementation G\vast spots incorrect behaviour rather quickly. G\vast shows the sequence of specification states, applied inputs and observed outputs of the SUT. A typical trace indicating behaviour not allowed by the specification is:

**Issue found! Trace:**

**SpecificationStates Input \rightarrow ObservedOutput**

1: [[ ]] (EnQ False) \rightarrow []
2: [[False]] DeQ \rightarrow [Just False]
3: [[ ]] isEmptyQ \rightarrow [True]
4: [[False]] (EnQ False) \rightarrow []
5: [[False]] Head \rightarrow [Nothing]

Allowed outputs and target states: [Pt [Just False] [False]]

These results show that it properties are quite powerful in detecting errors. Nevertheless there are incorrect implementations possible that passes the test with properties. Here the incorrect queue implementation has a finite maximum queue size. This is not covered by our properties. It is not difficult to add a test that covers this.

\[
\begin{align*}
p5 & : : \text{T} \rightarrow \text{Bool} \\
p5 \ l &= \text{deQueueElements} \ (\text{foldr} \ \text{enqueue} \ \text{newQueue} \ l) = l \\
\text{where}
\end{align*}
\]

dеQueueElements :: (Queue a) \rightarrow [a]
dеQueueElements q
| isEmptyQ q = []
| = case dequeue q of

(Just a, q1) = [a: deQueueElements q1]
(Nothing, _) = abort "there should be an element in the queue"

Otherwise the properties and the state machine are equal powerful in detecting errors. It is less work to define this single state machine based specification than the set of logical based properties. In our experience this holds for very many systems.

4 Constrained Datatypes

In our queue example the generation of test data was relative easy. For the property that needed queue, p4, we used generic generation since every instance of that type is a valid queue. Often we use data types that impose some additional constraints on its instances. For instance we use a data type that can represent any type to represent search trees without duplicates.

:: Tree a = Node a (Tree a) (Tree a) | Leaf

The abstract type for search trees based on this data type looks like:

newTree
isEmptyTree :: (Tree a) → Bool
insertTree :: a (Tree a) → Tree a | Ord a
elemTree :: a (Tree a) → Bool | Eq, Ord a
deleteTree :: a (Tree a) → Tree a | Ord a
inorder :: (Tree a) → [a]

Using this interface we can define a check to see if a tree is a valid search tree using the function inorder that lists the tree elements in an inorder way. The inorder sequence of elements of a search tree is sorted and contains no duplicates.

isSearchTree :: ((Tree a) → Bool) | Ord a
isSearchTree = valid o inorder

where
valid [a:b:r] = a < b && valid [b:r]
valid _ = True

As discussed in the previous section we use here characters as tree elements in the tests. Using this predicate we can generically generate arbitrary trees, by derive ggen Tree, select the search trees and express the desired property. For instance:

:: TreeT := Tree T
:: T := Char

ps1 :: T TreeT → Property
ps1 e t = isSearchTree t ⇒⇒ not (elemTree e (deleteTree e t))

In this test the fast majority of test cases will be rejected. The larger the number of test is that we want, the greater the fraction of test cases is that is rejected. Even if we generate only 20 test cases 60% of the test cases is rejected. In order
to prevent a false picture of the quality of the tests, \( \forall \)st counts the rejected arguments and the actual tests separately. In this example the test system reports: \textbf{Passed: maximum number of arguments (20) generated after 8 tests 12 cases rejected.}

We would obtain much more effective tests if only valid search trees would be used. Even without breaking the abstract tree interface this can be done by using the functions to the interface to construct search trees by defining:

\[
ggen[\text{TreeT}] n r = \text{map} (\text{foldr insertTree Leaf}) (ggen[\times] n r)
\]

If desired we can check whether the generated trees are indeed valid search trees by testing the property \( ps2 \):

\[
ps2 :: \text{TreeT} \rightarrow \text{Bool}
ps2 t = \text{isSearchTree} t
\]

To get an impression of the test suite use we can label the test by size of the trees used. The property is changed to:

\[
ps2l :: \text{TreeT} \rightarrow \text{Property}
ps2l t = \text{label} (\text{sizeTree} t) (\text{isSearchTree} t)
\]

The test result is:

"ps2l" Passed after 1000 tests
0: 1 (0.1%)
1: 47 (4.7%)
2: 217 (21.7%)
3: 26 (2.6%)
4: 11 (1.1%)
5: 188 (18.8%)
6: 510 (51%)

Which indicates that this test suite is much more effective that to one by selecting the search trees from the generated trees.

We observed that the size of the data types is much more important for the issue catching quality of the test than the elements stored in the data type. We are experimenting with various variation of the generic generation algorithm that give some preference to the recursive calls. More details will be available at TFP and in the final paper.

5 Conclusion

Although an automatic test system tests properties automatically, we can influence the quality and effectiveness of these tests by relative simple means. Some of them have to be done by the test engineer, others can be done by an improved implementation of the test system.
References

Restricted Function Patterns

Gergely Dévai

Eötvös Loránd University, Budapest, Hungary
Fac. of Informatics, Dept. of Programming Languages and Compilers
deva@elte.hu

Abstract. Users of embedded languages might want to pattern match on embedded programs. Making this possible requires a considerable effort from the developer of the language, because the underlying data types are usually hidden.
This paper first analyses the available solutions for this problem. As pattern synonyms [12] and function patterns [5] seem promising, a compromise between these two is proposed: restricted function patterns. These are more general than pattern synonyms, but it is still possible to process them at compile time. It is interesting that this proposal makes Haskell's exceptional rules about matching numeric literals more regular.
Finally, a lightweight prototype implementation is presented, that implements the functionality of the proposal, but cannot give the static guarantees that proper compiler support could achieve.

Keywords: function patterns, pattern matching, embedded languages

1 Problem definition

It is usually desirable to hide the implementation details of libraries and provide abstract interfaces for the users. On the other hand, this prevents the user from performing pattern matching, a convenient feature of functional languages. There are numerous proposals to tackle this old problem. This paper will definitely not try to solve this issue in general, but will inspect it from the point of view of embedded languages and explore the possibility of using a special class of functions in patterns.

Let us start with a toy embedding which in turn is a suitable model to study problems to be solved when creating embedded languages.

data Expr = Symbol String | Expr :: Expr

Function symbols and values are represented by their names via the Symbol constructor, while application (:$) can be used to build compound expressions by applying a function expression to an argument. Using this simple type one can already define basic arithmetic operations and start manipulating arithmetic expressions. The most convenient way to do this in Haskell is the instantiation of the Num class.

* This research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).
instance Num Expr where
  fromInteger n = Symbol $ show n
  a + b = Symbol "+" :$ a :$ b
  a - b = Symbol "-" :$ a :$ b
  a * b = Symbol "*" :$ a :$ b
  abs a = Symbol "abs" :$ a
  signum a = Symbol "signum" :$ a

Using the terminology of language embedding, the Expr type is used to build
the abstract syntax tree of embedded programs, while the Num instance serves
as the user interface and defines a piece of the syntax. It seems like a good idea
to hide the internal representation, i.e. the constructors of the Expr type behind
a module boundary and only expose the functions in the Num class to the user.

Let us suppose that one would like to use this language to optimize expres-
sions based on arithmetic laws:

\[
\begin{align*}
  0 + a &= a \\
  1 * a &= a \\
  0 * a &= 0
\end{align*}
\]

The desirable implementation of this would use pattern matching of the following
form:

\[
\begin{align*}
  \text{optimize} :: \text{Expr} & \rightarrow \text{Expr} \\
  \text{optimize} (0 + a) &= a \\
  \text{optimize} (1 * a) &= a \\
  \text{optimize} (0 * a) &= 0 \\
  \text{optimize} a &= a
\end{align*}
\]

However, this is invalid as the left-hand sides of these equations are not patterns.
In fact, without further support from the library, it is impossible to implement
this transformation, because the interface provides functions only for construct-
ing, but not for deconstructing expressions.

The next section briefly summarizes the currently available techniques. It
concludes that pattern synonyms and function patterns are quite close to what
we want to achieve here. Based on these, section 3 defines restricted function
patterns. Section 4 addresses the difficulties related to the proposal. Section
5 shows that the discussed extension makes Haskell’s pattern matching more
regular. Finally, the last two sections present a library to test the functionality
of the extended pattern matching and conclude the paper.

2 Available solutions

2.1 Selectors

One way to make deconstruction of expressions possible while hiding its con-
structors is to provide the user with a set of functions to examine expressions
and ask for their parts. These functions fall into two categories: some of them provide information on the form of the expression and thus can be used to separate cases, other functions return components of an expression of a given form.

```haskell
add :: Expr -> Bool
add (Symbol "+" :$ _ :$ _) = True
add _ = False

mul :: Expr -> Bool
mul (Symbol "*" :$ _ :$ _) = True
mul _ = False

arg1 :: Expr -> Expr
arg1 (_ :$ a :$ _) = a

arg2 :: Expr -> Expr
arg2 (_ :$ _ :$ a) = a
```

The functions `add` and `mul` yield true if the expression in question is a compound one with topmost operation addition or multiplication respectively. Unpacking the arguments of such expressions are done by selectors like `arg1` and `arg2`. Using these functions one can implement the desired optimization transformation:

```haskell
optimize :: Expr -> Expr
optimize e
  | add e && arg1 e == 0 = arg2 e
  | mul e && arg1 e == 1 = arg2 e
  | mul e && arg1 e == 0 = 0
  | otherwise = e
```

This approach is problematic for the following reasons:

- A considerable amount of utility functions must be provided to make deconstruction of expressions convenient.
- Partial functions like `arg1` and `arg2` in our example makes the library dangerous.
- Implementation of the optimization transformation is far from the clarity of the desirable solution envisioned at the end of section 1.

### 2.2 Data type for matching

This solution enables pattern matching by adding a data type to the interface of the library. A function to convert expressions to this additional type is also provided.

```haskell
data Arith
  = Expr :+: Expr
  | Expr :-: Expr
```

311
One can first use the `arith` function to convert an expression to an arithmetic expression. This transformation may fail, for example if the topmost operation of the expression is not one of the selected arithmetic functions. In this case the result is `Other`. Pattern matching can distinguish between failure and success as well as examine the topmost operation and access its arguments in the latter case.

There are two possibilities to construct the `Arith` type:

- If it is a recursive data type (i.e. the constructor parameters are of type `Arith`), pattern matching can analyze the structure in depth, but the extracted arithmetic expressions need to be converted back to `Expr`. This means that an inverse conversion is needed in addition.
- In the setting above (i.e. the constructor parameters are of type `Expr`) pattern matching is limited to one level at a time, but the arguments extracted are already of type `Expr`. This saves us from writing the inverse conversion. Note that the limitation mentioned here is elegantly removed by the `view patterns` extension described in section 2.3.

Using this solution one gets the following implementation of our model transformation:

```haskell
optimize :: Expr -> Expr
optimize e = case arith e of
  0 :+: a -> a
  1 :*: a -> a
  0 :*: a -> 0
  _ -> e
```

This solution is much more satisfactory compared to that of section 2.1, but still not perfect:

- Arithmetic operations are represented in a different way in patterns (` :+:`) and expressions (`+`).
- There is still a considerable overhead when writing the library: extra data type(s) and conversion function(s) are needed to make pattern matching possible.
2.3 View patterns

This is a light-weight Haskell extension [3] that goes well with the solution of the previous section. The conversion from \texttt{Expr} to \texttt{Arith} can happen inside the pattern:

\begin{verbatim}
optimize :: Expr -> Expr
optimize (arith -> 0 :+: a) = a
optimize (arith -> 1 :*: a) = a
optimize (arith -> 0 :*: a) = 0
optimize e = e
\end{verbatim}

To match an expression with a view pattern, the value is first transformed using the function before the arrow and the result is matched with the pattern on its right hand side. The real power of this extension is shown when view patterns are nested. One can write patterns like the one in this function:

\begin{verbatim}
distr (arith -> a :*: (arith -> b :+: c)) = a * b + a * c
\end{verbatim}

Similar solution can be achieved with transformational patterns described in [11]. These extensions sometimes make the solution of section 2.2 more elegant, but the problems listed there still apply.

2.4 Active patterns

The data type of the toy embedding from section 1 looks like this in the F# language:

\begin{verbatim}
type Expr = Symbol of string | App of Expr * Expr
\end{verbatim}

Instead of view patterns, there is a feature called \textit{active patterns} [16] in F#. This language element merges the data type to be used for matching and the conversion function into one artifact:

\begin{verbatim}
let (|Zero|One|Add|Mul|Other|) e =
  match e with
  | Symbol "0" -> Zero
  | Symbol "1" -> One
  | App(App(Symbol "+", a), b) -> Add(a,b)
  | App(App(Symbol "*", a), b) -> Mul(a,b)
  | _ -> Other
\end{verbatim}

The reason for having \texttt{Zero} and \texttt{One} in the active pattern is that numeric literals are not polymorphic in F#: 0 is just an int and cannot have type \texttt{Expr}. For the same reason we need a simple function encoding 0 of \texttt{Expr}:

\begin{verbatim}
let zero = Symbol "0"
\end{verbatim}

Now we can formulate the optimization function using pattern matching with active patterns:
let opt e =
match e with
| Add(Zero,a) -> a
| Mul(One,a) -> a
| Mul(Zero,_) -> zero
| _ -> e

At first sight this is much less elegant compared to the pattern matching solutions presented so far. But let us imagine that F# had polymorphic numeric literals and supported operators in active patterns. Instead of Zero and zero we could have 0, and instead of Add and Mul something like :+: and :*: respectively. The result would be even closer to the optimal than solutions of the previous two sections, because explicit application of the conversion function is not needed.

Yet, this does not solve the root of the problem: some extra work (definition of the active pattern) is needed to make pattern matching possible.

### 2.5 Pattern synonyms

The *Strathclyde Haskell Enhancement* [12] is a preprocessor for Haskell that implements a selection of proposed language extensions. One of them is called pattern synonyms. One can define pattern synonyms using the following syntax:

```haskell
pattern Add x y = Symbol "+" <$> x <$> y
pattern Mul x y = Symbol "+" <$> x <$> y
```

Add and Mul are allowed to appear both in expressions and in patterns. They are simply replaced with the right-hand sides of their definitions much like macros.

The idea goes back to abstract value constructors introduced in [4]. To make semantics clear and the replacement possible at compile time, strict rules constrain the definition of pattern synonyms. The names are capitalized identifiers and the right-hand sides have to be valid patterns except that pattern synonyms are also allowed in place of constructors. Consequently, they are linear: all variables are used exactly once in the right-hand side of the definition. However, the currently available implementation does not check this property.

Using the pattern synonyms defined above, our optimization function takes this form:

```haskell
optimize :: Expr -> Expr
optimize (Add 0 e) = e
optimize (Mul 1 e) = e
optimize (Mul 0 e) = 0
optimize e = e
```

The result is similar to that of section 2.4 and could be even nicer if infix pattern synonyms were supported. So far this solution requires the least additional effort from the programmer of the library. It may sometimes even be possible to provide pattern synonyms instead of functions in the public interface, but the limitations on the form of identifiers may make this undesirable and it is sometimes (like in our case with the functions of the Num class) impossible.
2.6 Function patterns

Curry [10] is a functional logic programming language based on Haskell and Prolog. Its syntax is similar to that of Haskell, therefore the definition of the `Expr` data type seen in section 1 is completely valid Curry code. Unfortunately, Curry does not support type classes and numeric literals are not polymorphic. This implies the following changes in the front-end of our toy language embedding:

```
num :: Int -> Expr
num n = Symbol $ show n

(+) :: Expr -> Expr -> Expr
x +. y = Symbol "+" :$ x :$ y

(*) :: Expr -> Expr -> Expr
x *. y = Symbol "*" :$ x :$ y
```

The reason for mentioning Curry in this paper is that it provides the best solution for the problem at hand: function patterns [5]. Patterns in Curry are allowed to contain not only constructors and variables, but also functions, including built-in and user-defined ones.

```
optimize :: Expr -> Expr
optimize (num 0 +. x) = x
optimize (num 1 *. x) = x
optimize (num 0 *. _) = num 0
optimize e = e
```

If it was possible to overload arithmetic operations and numeric literals in Curry, this definition would be exactly our wish. Note that this is achieved without any further support from the library, because the functions used to construct expressions can also be used in patterns to deconstruct them.

However, the semantics of this extended pattern matching in Curry is far from that of Haskell. For example, the function

```
f :: [Int] -> ([Int], [Int])
f (xs ++ ys) = (xs, ys)
```

has multiple results for nonempty lists. The function application `f [1,2]` yields the following set of results: (`([], [1,2])`, `([1], [2])`, `([1,2], [])`). This is acceptable in a functional logic language, but unsuitable for languages like Haskell. Furthermore, function patterns of Curry are transformed to traditional patterns at runtime.

3 Restricted function patterns

The solutions presented in the previous section can be divided into the following categories:
Section 2.1 avoids pattern matching. It is inconvenient and unsafe.

Sections 2.2-2.4 are all approximations of views [17]. The basic idea behind them is that the value to be matched is first transformed to something that can be matched. The problem here is that, in addition to implementing functions to construct entities of the embedded language, a considerable effort is needed to provide ways of deconstructing them via pattern matching.

An orthogonal possibility is shown in sections 2.5 and 2.6. Both pattern synonyms and function patterns provide a way to build patterns that we would like to match some value with.

The goal here is to define a restricted class of function patterns that are more powerful than pattern synonyms, but much more restricted than Curry’s function patterns in order to provide static guarantees and no performance overhead. The idea is to allow arbitrary expressions in patterns, if there is an equivalent valid (traditional) pattern.

**Definition 1.** If functions \( f \, x_1 \, x_2 \, \ldots \, x_n = e \) and \( f' \, x_1 \, x_2 \, \ldots \, x_n = e' \) are both of type \( A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n \rightarrow B \), such that they are (extensionally) equal and \( e' \) is a valid pattern, then \( e \) is a restricted function pattern (RFP), and \( e' \) is its canonical form.

For example, all patterns in the “desired implementation” of the optimize function presented at the end of section 1 are restricted function patterns. The following table shows their canonical forms.

<table>
<thead>
<tr>
<th>RFP</th>
<th>Canonical form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0+a</td>
<td>Symbol &quot;+&quot;: Symbol &quot;0&quot; : Symbol &quot;a&quot;</td>
</tr>
<tr>
<td>1*a</td>
<td>Symbol &quot;*&quot;: Symbol &quot;1&quot; : Symbol &quot;a&quot;</td>
</tr>
<tr>
<td>0*a</td>
<td>Symbol &quot;*&quot;: Symbol &quot;0&quot; : Symbol &quot;a&quot;</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

Is a given expression RFP or not? This is not a decidable problem in general as results about the halting problem show. Nevertheless, it is possible to construct a decision algorithm that accepts only RFPs. It rejects all non-RFPs, but also some of the RFPs. For example, symbolic execution of the expression with a fixed limit on the reduction steps is such an algorithm. An expression is accepted if symbolic execution reaches the canonical form. If it is impossible to continue symbolic execution (for example non-trivial pattern matching is performed on a parameter) or the number of reduction steps reach the limit, the function is rejected.

This decision algorithm also provides the canonical form of all accepted RFPs. If we allow RFPs in pattern matching, the compiler can simply replace them by the corresponding canonical form. As those are “normal” patterns, it is possible to process them in the usual way. Note that all this happens in compile time: an RFP is transformed to an ordinary pattern making no runtime overhead and no change to the pattern matching algorithm.

In fact, a smart compiler already does something similar for optimization purposes. It may perform symbolic computation to reduce the expressions as much
as possible to increase runtime performance. RFPs are good candidates for such an optimization. The difference is that the optimization-related transformations happen in expressions instead of patterns.

4 Problems and solutions

4.1 Lexical ambiguity

Haskell patterns contain constructors and variables. Constructors are capitalized identifiers or operators starting with a colon, while variables are identifiers starting with a lower-case letter. In contrast, RFPs can contain ordinary functions that are syntactically indistinguishable from variables.

\[
x :: \text{Int} \\
x = 5
\]

\[
f :: \text{[Int]} \rightarrow \text{Int} \\
f [] = 0 \\
f (x:xs) = x + f xs
\]

In this example it is not clear if \(x\) in the pattern \((x:xs)\) is the zero-arity function with value 5 or a free variable of the pattern. Note that this ambiguity is only present in case of symbols without arguments. In the RFP \((a \ b)\) the symbol \(a\) can only be a function.

The same issue is present in the Agda programming language [1] that allows lower-case constructors. Agda's rule says that \(x\) can only be a variable if it is not defined as a constructor. In contrast, Curry considers these symbols as variables and gives a warning if they shadow zero-arity functions. Another possible solution is to invent some syntax to distinguish variables from functions in dubious cases.

4.2 Type system issues

A much more important problem is that most of the type systems used in practice cannot express that an expression is an RFP. This is problematic, because changing the implementation of a function may make the compiler complain about patterns even if the type of the function is untouched. The following example illustrates the problem.

\[
double :: \text{Expr} \rightarrow \text{Expr} \\
double x = 2 \times x
\]

The expression \(\text{double x}\) is an RFP with canonical form \(\text{Function "\times" :$ Number 2 :$ x}\). Let us use it in the following function definition:

\[
halve :: \text{Expr} \rightarrow \text{Expr} \\
halve (\text{double x}) = x \\
halve x = \text{Symbol "div" :$ x :$ 2}
\]
What happens if we change the implementation of double slightly?

\[
\text{double :: Expr -> Expr}
\]
\[
\text{double } x = x + x
\]

The expression \(\text{double } x\) now reduces to \(\text{Symbol } * :\! x :\! x\), which is not a valid pattern any more, at least not in a language with linear patterns. The compiler in this case will reject the first equation of the \text{double} function. This may be annoying since the type of \text{double} did not change. Programmers of languages with powerful type systems got used to the fact that the type of a function in itself defines where it is allowed to appear in the program.

There are two possible solutions: either the type system is extended, or the compiler has to handle non-restricted function patterns in a different way. The rest of this section explores these possibilities.

**Extending the type system** If an expression is well-defined, parametric and linear, then it is an RFP. All of these properties are also important from purposes different from this paper, and there are type systems created to ensure them.

**Well-definedness** If the function \(x_1 x_2 \ldots x_n \rightarrow e\) is not well defined for a set of (well-defined) arguments \(v_1 v_2 \ldots v_n\), then \(e\) cannot be an RFP. Raising an exception, returning \textit{undefined} or going into infinite recursion are not allowed. Type theory knows about many different type systems that ensure termination of expressions.

**Parametricity** Parametric functions use their parameters as black boxes: there is no pattern matching allowed on them. For example, the \text{singleton} function below is parametric, but \text{length} is not.

\[
\text{singleton :: a -> [a]}
\]
\[
\text{singleton } x = [x]
\]

\[
\text{length :: [a] -> Int}
\]
\[
\text{length } [] = 0
\]
\[
\text{length } (x:xs) = 1 + \text{length } xs
\]

Parametric functions are important from our perspective, because this property ensures that symbolic execution is possible: it is possible to reduce the expression without fixing its free variables. Together with well-definedness, this ensures that symbolic execution terminates and leads to an expression containing constructors and variables only.

Parametric functions form a well-studied class that is important for language embedding also for a different reason: \textit{higher order abstract syntax} (HOAS) [14]. If one wanted to extend the \texttt{Expr} type from section 1 with a new constructor for \(\lambda\)-abstraction, an elegant solution would be this one: \texttt{Lam (Expr -> Expr)}. This makes nontrivial functions (like \(\beta\)-reduction, for example) quite easy to implement.
On the other hand, passing a non-parametric function to \texttt{Lam} results in an expression which has no corresponding \( \lambda \)-term, like in case of the following example.

\begin{verbatim}
Lam \$ \ x \rightarrow \ \text{case x of}
    \Symbol \ "a" \rightarrow \ \Symbol \ "a"
    \text{otherwise} \rightarrow \ \Symbol \ "b"
\end{verbatim}

A great amount of work has gone into solving these problems. An early one is an extension to the ML language \cite{13} that makes pattern matching possible on parametric functions. In \cite{6} a type system is developed that can distinguish the parametric and non-parametric function spaces. Such a type system would also be useful for checking RFPs.

\textit{Linearity} In many functional languages patterns must be linear: using the same variable more than once in a pattern is invalid. In such a language RFPs are also linear. Linear type systems \cite{18} are well studied and are important from the perspective of destructive updates and interaction with the real world while still keeping referential transparency. A notable example is the Clean language that implements uniqueness typing \cite{15} for this purpose.

\textit{Warnings instead of errors} If the type system is not strong enough to express that an expression is RFP, the compiler should not reject non-RFP patterns for the reason discussed at the beginning this section. So what should the compiler do if it finds a non-RFP pattern (or it is not able to prove that it is an RFP)? A possibility is to give a warning to the user and replace the pattern with one that fails to match any value. This way the decision of the compiler to accept or reject a pattern will only depend on the types of the functions involved. Changing their implementation (without touching their types) may affect only the runtime behavior of the pattern match and may result in warnings.

5 A closer look at the \texttt{Num} class

One of the ingenious features of Haskell that makes this language particularly suitable for language embedding is that numeric literals are polymorphic. The literal \texttt{0} for example can be of any type that implements the \texttt{Num} class. In case of our toy embedding, the literal \texttt{0} of type \texttt{Expr} means \texttt{Symbol \ "0"}, because of the implementation of the \texttt{fromInteger} function for the \texttt{Expr} type. In fact, an integer literal \texttt{n} is automatically transformed to \texttt{fromInteger n} by the compiler.

The nasty bit is that the same has to happen also in patterns. In the \texttt{Num} instance below, the \texttt{fromInteger} function is \texttt{undefined}, and this makes pattern matching unexpectedly unsafe.

\begin{verbatim}
data Foo = Foo
deriving (Eq, Show)
\end{verbatim}
instance Num Foo where
  fromInteger _ = undefined

f :: Foo -> Bool
f 0 = False
f _ = True

Evaluating the expression \( f \text{ Foo} \) in the Haskell interpreter yields an exception, because the \texttt{fromInteger} function is called to transform the literal 0 in the pattern to type Foo. If the implementation is changed to

\[
\text{fromInteger } n \text{ = fromInteger } n
\]

then \( f \text{ Foo} \) goes into infinite recursion.

What is particularly strange here is that \texttt{undefined} and infinite recursion is the result of processing the pattern and not the value that is matched. Moreover, there is no guarantee that the result of \texttt{fromInteger } \( n \) is a valid pattern, therefore equality is used instead of normal pattern matching. For this reason, in order to be an instance of the \texttt{Num} class, the type \texttt{Foo} also has to have an \texttt{Eq} instance. In the implementation above this is fulfilled by the \textit{deriving close} at the end of the data type definition.

Now let us correct the implementation of \texttt{fromInteger} and ruin that of equality.

instance Num Foo where
  fromInteger _ = Foo

instance Eq Foo where
  (==) = undefined

The result of \( f \text{ Foo} \) is an exception again.

Is it possible to correct this? As discussed above, the compiler calls the function \texttt{fromInteger} to convert integer literals to the desired type. Let us write this conversion explicitly in the pattern even if this is incorrect Haskell code:

f :: Foo -> Bool
f (fromInteger 0) = False
f _ = True

Now it is clear that this is a generalization of the pattern language very much related to what we discuss in this paper.

- If \texttt{fromInteger 0} is an RFP (like in case of \texttt{Expr} or the built-in numeric types), then using the literal 0 is all right: \texttt{fromInteger 0} should be replaced by its canonical form and the usual pattern matching algorithm can be used instead of equality. This way there is no need to make \texttt{Eq} a superclass of \texttt{Num}.
- If \texttt{fromInteger 0} is not an RFP (for example if it is \texttt{undefined}), then the compiler should complain about the pattern according to section 4.2.
This means that the RFP extension discussed in this paper would make Haskell’s current pattern matching rules related to numeric literals more consistent and safer.

6 Implementation

Before creating a proper language extension, there is more research needed about the issues discussed in section 4. Nevertheless, it was already possible to create a lightweight Haskell library that provides the functionality of restricted function patterns, even so it is not able to check the restrictions at compile time and cannot guarantee the same run-time performance that built-in compiler support could. This library is currently available for testing at [7] and is expected to be uploaded to the Hackage library database [2].

6.1 Public interface

Using this library one can implement the optimization function discussed in section 2 as follows.

```haskell
opt :: Expr -> Expr
opt e = match e $ do
    with $ \a -> 0 + a ~> a
    with $ \a -> 1 * a ~> a
    with $ \a -> 0 * a ~> 0
    with $ \a -> a ~> a
```

The match function gets two arguments: the value to be matched and a sequence of cases. In order to mimic the syntax of Haskell’s case expressions, the cases are listed in a monadic environment (hence the do keyword). Cases are created using the with function of arity one. Its argument is a function with arbitrary number of arguments (including zero) and it produces a pattern and a corresponding result combined by the ( ~> ) operator.

If only RFPs are used as patterns, the library guarantees that a match is equivalent to a case expression with the corresponding normal forms. However, the library is not able to check at compile time if the patterns are really RFPs: this is the responsibility of the user.

- If the pattern is not well-defined (due to undefined or non-termination), the match may fail, terminate with undefined or hang.
- If the pattern is not parametric, the match may fail, succeed or (in most of the cases) the nonparametric pattern error is raised.
- If the pattern is not linear, the match may fail or the nonlinear pattern error is raised.

It might be disappointing that non-parametric patterns makes matching so unpredictable, but it is important to note that a successful match is always correct.
Whenever a value \( v \) matches a case \( \{ x_1 x_2 \ldots x_n \rightarrow p \rightleftharpoons e \) and the variables get bound to the values \( v_1, v_2, \ldots, v_n \) respectively, the value \( (\{ x_1 x_2 \ldots x_n \rightarrow p \}) v_1 v_2 \ldots v_n \) is well-defined and really matches \( v \). This also holds for erroneous patterns.

In order to make this kind of pattern matching possible for a new data type, one has to make it an instance of the Matchable class. This instance defines the components of compound values. For example, the Matchable instance of the Expr type is the following.

```haskell
instance Matchable Expr where
  Symbol s .=. Symbol z = Just [s :=: z]
  (e :$ f) .=. (g :$ h) = Just [e :=: g, f :=: h]
  _ .=. _ = Nothing
```

This means that
- symbols match, if their names match,
- function applications match, if both the functions and the arguments match,
- there is no other way for expressions to match.

Matchable instances are trivial and could be automatically created by the compiler.

### 6.2 Pattern matching algorithm

Matching a value \( v \) with a case \( c = \{ x_1 x_2 \ldots x_n \rightarrow p \rightleftharpoons e \) is performed as follows. First, the system creates parameters of two kinds that we denote by \( T \) and \( F \). The function \( c \) is applied to these parameters such that \( x_1 \) becomes \( T \) and all others from \( x_2 \) to \( x_n \) become \( F \). Let us denote the result by \( p^* \rightleftharpoons e^* \). Now \( p^* \) is matched with \( v \) using an algorithm described later below and this leads to one of the following results:

- If the match fails, the next case is tried.
- If the match succeeds, it provides a value \( v_1 \) that \( x_1 \) will be bound to. The binding is done by applying \( c \) to \( v_1 \). This yields the function \( c_1 = \{ x_2 \ldots x_n \rightarrow p_1 \rightleftharpoons e_1 \). Then the whole algorithm is called again recursively with this reduced case that has one argument less than the original.

After eliminating all arguments without failure, one gets \( p_n \rightleftharpoons e_n \). If \( p_n \) matches \( v \), then the case fires and the result is \( e_n \).

Now let us describe how a pattern \( pat \) is matched with a value \( val \):

- If \( pat \) is the parameter \( T \) then \( val \) is returned as a candidate for the next binding.
- If \( pat \) is the parameter \( F \) then the match succeeds, but no value is returned.
- If \( pat \) is not a parameter then \( val . = . m \) is called (see the Matchable class above). It provides a list of match conditions, and matching is done recursively for all pairs in this list:
• If any of them fail, the whole match fails.
• If all of them succeed and either none of them or at least two of them return a value, then the pattern is not linear and an error is raised.
• If all of them succeed and exactly one returns a value, then the match succeeds and returns the value.

Matching $p_n$ with $v$ is done similarly, but in this case no value is returned (as there are no more variables to bind).

What are the parameters $T$ and $F$ technically? Besides the function $(\cdot = \cdot)$ seen before, the `Matchable` class contains two more functions:

```haskell
makeParam :: Bool -> a
isParam :: a -> Maybe Bool
```

Parameter $T$ and $F$ are `makeParam True` and `makeParam False` respectively. The `isParam` function decides if its argument is a parameter or not. The following rules must hold:

- $\text{isParam} (\text{makeParam True}) == \text{Just True}$
- $\text{isParam} (\text{makeParam False}) == \text{Just False}$
- If $a$ is not a parameter, then $\text{isParam} a == \text{Nothing}$.

Users of the library do not have to implement these functions (unless they want to). The default implementation of `makeParam` throws an exception that wraps its argument. `isParam` tries to evaluate its argument to head normal form. If this succeeds then it was not a parameter, otherwise the exception is caught and the wrapped logical value is returned. This is only possible within the IO monad, but it is hidden using the `unsafePerformIO` function. One reason for writing custom implementations of `makeParam` and `isParam` may be that one works with exceptions interfering with the ones used by the library.

### 7 Conclusion

Embedded languages and similar functional libraries often hides the constructors of their data types. Instead they provide functions in their public interfaces that can be used for data construction. This design makes pattern matching in user code impossible. To make it possible after all, the library has to provide additional tools to deconstruct data. This paper searches ways to avoid (or considerably reduce) this additional work.

Pattern synonyms and function patterns are promising techniques. While the former one is two restrictive, the latter one is too general for a functional programming language. Therefore this paper proposes a compromise between the two; restricted function patterns. A class of expressions is defined that is safe to be allowed in patterns, because they can be replaced with equivalent traditional patterns at compilation time.

Problems related to this language extension are analyzed and solutions are proposed. Finally, a lightweight implementation is presented that provides the functionality of restricted function patterns. The author of this paper used this library in two embedded language projects [9,8].
References

1. Agda programming language.  
   http://wiki.portal.chalmers.se/agda/.
2. HackageDB.  
   http://hackage.haskell.org/.
7. Gergely Dévai. Restricted Function Patterns (Haskell library).  
   http://deva.web.elte.hu/FunPat/.
   http://personal.cis.strath.ac.uk/~conor/pub/she/.
Applicative Shortcut Fusion

Germán Andrés Delbianco\textsuperscript{1}, Mauro Jaskelioff\textsuperscript{2}, and Alberto Pardo\textsuperscript{3}

\textsuperscript{1} IMDEA Software Institute, Spain
\textsuperscript{2} CIFASIS-CONICET/Universidad Nacional de Rosario, Argentina
\textsuperscript{3} InCo, Universidad de la República, Uruguay

Abstract. In functional programming one usually writes programs as the composition of simpler functions. Consequently, the result of a function might be generated only to be consumed immediately by another function. This potential source of inefficiency can often be eliminated using a technique called shortcut fusion, which fuses both functions involved in a composition to yield a monolithic one. In this article we look at shortcut fusion for applicative computations. Applicative functors provide a model of computational effects which generalise monads, but they favour an applicative programming style. To the best of our knowledge, this is the first time shortcut fusion is analysed in an applicative setting.

1 Introduction

One of functional programming much advocated benefits is the possibility of easily constructing large and complex programs through the combination of smaller or simpler ones [12]. This modular approach, however, often results in programs which are quite inefficient when compared to their monolithic counterparts: compositional design often involves creating an intermediate data structure which is immediately consumed. In order to ameliorate this problem, several formal techniques have been developed that allow the derivation of efficient programs from simpler modular ones.

Among these techniques lies shortcut fusion [11,19] which is concerned with the elimination of unnecessary list traversals. It is based on a single transformation: the foldr/build rule which fuses the application of a uniform list-consuming function, expressed as a fold on lists, to the result of a uniform list-generating function, expressed in terms of the build combinator. This fusion rule can be generalised to any inductive datatype, yielding the following generic rule:

\[
fold k \circ build \ g = g \ k
\] (1)

Shortcut fusion has been extended to cope with cases where the intermediate structure is produced in certain contexts. For example, shortcut fusion has been considered for monadic computations [7,13,14], unstructured functors [8], accumulations [15] and circular programs [6,18].
A recent development is the notion of applicative functor [16]. Applicative functors provide a novel manner in which effectful computations can be constructed that has gained a rapid acceptance among functional programmers. However, shortcut fusion under an applicative context has not yet been studied. Precisely, in this article, we investigate shortcut fusion under the context of an applicative computation, and analyse manners in which to take advantage of some characteristic features of applicative programming. The contributions of this article are:

- We provide a shortcut fusion rule for applicative computations. The rule shows the importance and generality of \textit{traverse} for generating applicative structures.
- We provide two combinators, \textit{ifold} and \textit{ibuild} which model the uniform consumption and production of applicative computations.

The paper is organised as follows. In Section 2 we present some preliminaries such as the representation of datatypes and some fusion laws. In Section 3 we review the concepts of applicative and traversable functors. In Section 4 we introduce our applicative shortcut fusion law. Finally, in Section 5 we conclude and discuss future work.

2 Data type theory

In this section, we review concepts about datatypes, recursive program schemes, and some associated transformation laws that are necessary throughout the rest of this paper. The constructions to be presented are datatype-generic \cite{2,3,9} in the sense that they are valid for a wide class of datatypes.

2.1 Data types

The structure of data types can be captured using the concept of a \textit{functor}. A functor consists of a type constructor \( f \) and a map function:

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

where \( \text{fmap} \) must preserve identities and compositions: \( \text{fmap} \ id = id \) and \( \text{fmap} \ (f \circ g) = \text{fmap} \ f \circ \text{fmap} \ g \). A standard example of a functor is that formed by the list type constructor and the well-known \textit{map} function.

Recursive data types correspond to least fixed points of functors. Given a data type declaration it is possible to derive a functor \( f \), which captures the structure of the type, such that the data type can be seen as the least solution of the equation \( x \equiv fx \) \cite{1}. In Haskell, we can encode this isomorphism defining a type constructor \( \mu :: (\* \rightarrow \*) \rightarrow \* \) as follows:

```haskell
newtype \( \mu \) \( f \) = In { \text{unIn} :: f (\mu f) }
```
Example 1 (Naturals). Given a data type for natural numbers,

\[
\textbf{data} \ \text{Nat} = \text{Zero} \mid \text{Succ Nat}
\]

its signature is given by a functor FN defined as follows:

\[
\begin{align*}
\textbf{data} \ FN \ x &= \text{FZero} \mid \text{FSucc} \ x \\
\textbf{instance} \ \text{Functor} \ FN \ \text{where} \\
\text{fmap} \ f \ \text{FZero} &= \text{FZ} \\
\text{fmap} \ f \ (\text{FSucc} \ n) &= S \ (f \ n)
\end{align*}
\]

So, alternatively, we can say that \( \text{Nat} = \mu FN \).

For polymorphic types, it is necessary to use functors on multiple arguments to capture their signature. This is because of the presence of the type parameters. For example, for types with one parameter we need a functor on two arguments, usually called a \textit{bifunctor}, to represent their structure.

\[
\textbf{class} \ \text{Bifunctor} \ f \ \text{where} \\
\text{bimap} :: (a \to b) \to (c \to d) \to f \ a \ c \to f \ b \ d
\]

Example 2 (Lists). The structure of polymorphic lists, \([a]\), is captured by a bifunctor FL,

\[
\begin{align*}
\textbf{data} \ FL \ a \ b &= \text{FNil} \mid \text{FCons} \ a \ b \\
\textbf{instance} \ \text{Bifunctor} \ FL \ \text{where} \\
\text{bimap} \ f \ g \ \text{FNil} &= \text{FNil} \\
\text{bimap} \ f \ g \ (\text{FCons} \ a \ b) &= \text{FCons} \ (f \ a) \ (g \ b)
\end{align*}
\]

By fixing the bifunctor argument corresponding to the type parameter \(a\) (the type of the list elements) we get a functor \(FL a\) which represents the signature of lists of type \(a\):

\[
\textbf{instance} \ \text{Functor} \ (FL \ a) \ \text{where} \\
\text{fmap} \ f \ \text{FNil} &= \text{FNil} \\
\text{fmap} \ f \ (\text{FCons} \ a \ b) &= \text{FCons} \ a \ (f \ b)
\]

Thus, \([a]\) = \(\mu (FL \ a)\).

2.2 Fold

Given a functor \(f\) that captures the signature of a data type and a function \(k :: f \ a \to a\) (called an \(f\)-algebra), we can define a program scheme, called \textit{fold} [3], which captures function definitions by structural recursion on the type \(\mu f\).

\[
\begin{align*}
\text{fold} :: \text{Functor} \ f \Rightarrow (f \ a \to a) \to \mu f \to a \\
\text{fold} \ k &= k \circ \text{fmap} \ (\text{fold} \ k) \circ \text{unIn}
\end{align*}
\]
The signature corresponding to a type $T$ with $n$ constructors is a functor that has also $n$ cases. The same occurs with the algebras for that functor; they are essentially a tuple $(k_1, \ldots, k_n)$ of $n$ component operations, each one with the appropriate type. For example, an algebra for the functor $FL a$ is a function $k :: FL a b \to b$ of the form:

- $k \text{ FNil} = e$
- $k (\text{FCons } a b) = f a b$

with components $e :: b$ and $f :: a \to b \to b$.

**Example 3 (Lists).** Fold for lists corresponds to the $\text{foldr}$ function, a well-known function pattern in functional programming [4]:

$$
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
$$

- $\text{foldr } f \ e \ [] = e$
- $\text{foldr } f \ e \ (x : xs) = f \ x \ (\text{foldr } f \ e \ xs)$

It traverses the list and replaces $[]$ by the constant $e$ and the occurrences of $(::)$ by function $f$. The replacement of the input datatype constructors by the corresponding operations of the algebra is a common characteristic of fold on any type.

Fold enjoys many algebraic laws that are useful for program transformation. One of them is *acid rain*, a *fusion law* for eliminating intermediate data structures generated in function compositions. Acid rain requires that both the consumer and producer of the intermediate data structure be expressible as a fold.

**Law 1 (fold-fold fusion)**

$$
\tau :: (\text{Functor } f, \text{Functor } g) \Rightarrow \forall a. (f \ a \to a) \to (g \ a \to a)
$$

$$
\Rightarrow
$$

$$
\text{fold } k \circ \text{fold } (\tau \ \text{In}) = \text{fold } (\tau \ k)
$$

The polymorphic function $\tau$ is usually called an *algebra transformer* from algebras of signature $f$ to algebras of signature $g$. Its role is to abstract the occurrences of the constructors of the intermediate datatype so that they can be replaced by the corresponding operations of the algebra $k$ of the consumer.

Later on it will be necessary to consider the following generalisation of the previous law. It states the case where the first fold generates the intermediate structure in the context of another structure represented by a functor. The external structure is then maintained by the law while the occurrences of the internal datatype are eliminated.

**Law 2 (generalised fold-fold fusion)**

$$
\tau :: (\text{Functor } f, \text{Functor } g, \text{Functor } h) \Rightarrow \forall a. (f \ a \to a) \to (g \ (h \ a) \to (h \ a))
$$

$$
\Rightarrow
$$

$$
\text{fmap } (\text{fold } k) \circ \text{fold } (\tau \ \text{In}) = \text{fold } (\tau \ k)
$$

328
2.3 Shortcut fusion

*Shortcut fusion* [11] is another technique for the elimination of intermediate data structures. It is a consequence of parametricity properties, known as “free theorems” [20], associated to polymorphic functions. Like fold-fold fusion, shortcut fusion requires the consumer to be expressible as a fold. The producer, on the other hand, is required to build the intermediate data structure using uniquely the constructors of the datatype. This is expressed in terms of a function, called $build$, which carries a “template” that abstracts from the function body the occurrences of those constructors. This requirement is similar to that of fold-fold fusion, but the body of the $build$ does not need to be a fold. As a consequence of this, shortcut fusion is able to capture more general cases than fold-fold fusion.

$$build :: (Functor f) \Rightarrow (\forall a. (f a \to a) \to c \to a) \to c \to \mu f$$

$$build g = g \text{ In}$$

Notice that the abstraction of the datatype’s constructors is represented in terms of a $f$-algebra. The idea of shortcut fusion is then to replace, in the producer, the occurrences of the abstracted constructors by corresponding operations in the algebra of the fold that appears as consumer. This transformation is usually referred to as the *fold/build law*.

**Law 3 (fold/build)**

$$fold k \circ build g = g k$$

Like in the case of fold-fold fusion, it is possible to formulate a generalised form of shortcut fusion which captures the case where the intermediate data structure is generated as part of another structure. This generalisation has been a fundamental tool for the formulation of shortcut fusion laws for monadic programs [14,8], and for the derivation of (monadic) circular and higher-order programs [18,6]. It is based on an extended form of build:

$$ebuild :: (Functor f, Functor h) \Rightarrow (\forall a. (f a \to a) \to c \to h a) \to c \to h (\mu f)$$

$$ebuild g = g \text{ In}$$

where $h$ is a functor that represents the structure in which the produced data type is contained. This is a natural extension of the standard build function. Using the extended build we can state the following law:

**Law 4 (extended fold/build)**

$$fmap (fold k) \circ ebuild g = g k$$

Again, like in Law 2, fusion acts on the occurrences of the internal structure, while the context structure is maintained unchanged.
3 Applicative Functors

An applicative functor [16] is a type constructor \( f :: \ast \to \ast \), equipped with two operations, presented in Haskell by means of the following type class:

\[
\text{class } (\text{Functor } f) \Rightarrow \text{Applicative } f \text{ where }
\]
\[
\text{pure} :: \ast \to f \ast \\
(\otimes) :: f (\ast \to \ast) \to f \ast \to f \ast
\]

Intuitively, \( \text{pure} \) lifts a pure computation into the effectful context defined by \( f \) and \( \otimes \) performs an effectful application. Instances of \( \text{pure} \) and \( \otimes \) must verify the applicative functor laws [16].

Monads are applicative functors, taking \( \otimes \) to be monadic application and \( \text{pure} \) to be \text{return}. Furthermore, there are applicative functors which are not monads. McBride and Paterson [16], present also two classes of non-monadic applicative functors: monoid accumulators and neparian functors. The former class consists of constant functors whose effect is to accumulate over a given monoid e.g. \( ([], (+ +), []) \), while the latter class implements a notion of datatype transposition or vectorisation.

3.1 Traversable Functors

An applicative action is a function of type \( a \to f b \) where \( f \) is an applicative functor. These applicative actions can be used to perform traversals over a certain class of data structures, threading an effect through the data structure. This class of data structures is called Traversable:

\[
\text{class } (\text{Functor } t) \Rightarrow \text{Traversable } t \text{ where }
\]
\[
\text{traverse} :: (\text{Applicative } f) \Rightarrow (a \to f b) \to t a \to f (t b)
\]

Alternatively, this class can be defined by means of a distributive law \( \text{dist} :: f (c a) \to c (f a) \) which pulls the effects out of the data structure. The function \( \text{dist} \) and \( \text{traverse} \) are interdefinable, with \( \text{dist} = \text{traverse id} \) and \( \text{traverse } \iota = \text{dist } \circ \text{fmap } \iota \). The latter definition gives a concise description of what an effectful traversal does: first populate the structure with effects by mapping the applicative action and then collect them with the distributive law. Although not every functor is Traversable, all regular functors and even some non-regular are, which makes them a quite vast family.

Example 4 (Lists). Polymorphic lists are Traversable, as witnessed by the following instance:

\[
\text{instance } \text{Traversable } [\ ] \text{ where }
\]
\[
\text{traverse } \iota [\ ] = \text{pure } [\ ] \\
\text{traverse } \iota (x : xs) = \text{pure } (:) \otimes \iota x \otimes \text{traverse } \iota xs
\]
It is possible to define \textit{datatype-generic} traversals for parametric data structures presented as fixpoint of the parametric bifunctors which capture their signatures, in the spirit of Section 2. In order to define \textit{traverse} generically, we must first establish when the signature of a datatype can be traversed:

\begin{verbatim}
class Bifunctor s ⇒ Bitraversable s where
  bitraverse :: (Applicative f) ⇒
              (a → f c) → (b → f d) → s a b → f (s c d)
\end{verbatim}

Gibbons and Oliveira \cite{Gibbons2010} present an equivalent characterisation: a bifunctor \( s \) is Bitraversable if for any applicative functor \( c \) there exists a natural transformation \( \text{bidist} :: s (c a) (c b) \to c (s a b) \) which serves as a distributive law between the signature bifunctor and the applicative functor. Such distributive law exists for any given regular datatype and it can be defined \textit{polytipically} i.e. by induction on the structure of the signature bifunctor \cite{Hirschberg2001, Gibbons2010}. As in the case of \textit{traverse} and \textit{dist} above, \textit{bitraverse} and \textit{bidist} are also interdefinable as \( \text{bidist} = \text{bitraverse} \ id \ id \) and \( \text{bitraverse} \ f \ g = \text{bidist} \circ \text{bimap} \ f \ g \). Thus, \textit{traverse} can be defined generically for all fixed points of Bitraversable functors.

\begin{verbatim}
traverse :: (Applicative f, Bitraversable s) ⇒
         (a → f b) → µ (s a) → f (µ (s b))
traverse ι = fold (fmap In ∘ bitraverse ι id)
\end{verbatim}

Gibbons and Oliveira \cite{Gibbons2010} also claim that the \textit{traverse} operator captures \textit{“the essence of the Iterator pattern”} and have studied some calculational properties of idiomatic traversals. In Section 4, we will show how traversals go yonder: they play a crucial role in the characterisation of applicative shortcut fusion.

\section{Structured Shortcut Fusion}

The shortcut fusion rule that we looked at the end of Section 2.3 is somewhat unsatisfactory since only takes into account the type constructor part of an applicative functor and ignores the additional structure available. In particular, the function passed to \textit{build} can construct the applicative computation ignoring the applicative interface.

In this section we aim to obtain a shortcut fusion rule which takes into account the way that applicative computations are constructed. This means obtaining a notion of \textit{applicative structural recursion} and of \textit{applicative build}. As we will show later, once we characterise what we mean by a well-behaved applicative computation, their definition will be straightforward.

\subsection{Applicative Structural Recursion}

We introduce a characterisation of \textit{applicative structural recursion} and propose an operator, \textit{ifold}, which captures this recursion pattern. The \textit{ifold} operator not only constitutes a powerful and flexible tool for structuring recursive algorithms
with applicative computational effects, but also provides a comprehensive set of rules for calculating efficient functional programs in the presence of applicative functors [5].

First, we will discuss why the obvious approaches do not work. Applicative structural recursion cannot be obtained simply by lifting fold:

$$\text{liftFold} :: (\text{Bifunctor } s, \text{Applicative } f) \Rightarrow f (s a b \rightarrow b) \rightarrow f (\mu (s a)) \rightarrow f b$$

$$\text{liftFold } \phi x = \text{pure } \text{fold} \odot \phi \odot x$$

In this case the only effects available are those at the top level. However, we would expect the effects to be interspersed between the inductive datatype. Again, considering a fold:: (s a (f b) \rightarrow f b) \rightarrow \mu (s a) \rightarrow f b is not structured enough. Neither the algebra nor the fold need to know about the applicative structure of f.

With obvious answers discarded, we have to roll up our sleeves and analyse applicative programs defined by structural recursion in order to determine which common patterns arise, if any. We will attempt to identify, in particular, what is the contribution of pure and \odot to this end, and how are pure values produced and consumed. We start by analysing a couple of examples with different applicative effects.

**Example 5 (Reciprocal List).** The Maybe monadic applicative functor models failure as a computational effect. We want to define a function which computes the sum of the reciprocals of a given list of numbers, failing if there is some 0 value in the input list. We can think the generation of the reciprocal of a value as an applicative action: if the value is nonzero a computation that produces its reciprocal is returned, else we fail via Nothing.

$$\text{recip} :: \text{Float} \rightarrow \text{Maybe } \text{Float}$$

$$\text{recip } x = \text{if } (x \neq 0) \text{ then } \text{pure } (1 / x) \text{ else Nothing}$$

We can use this applicative action to define sumrecips by structural recursion:

$$\text{sumrecips} :: [\text{Float}] \rightarrow \text{Maybe } \text{Float}$$

$$\text{sumrecips } [] = \text{pure } 0$$

$$\text{sumrecips } (x : xs) = \text{pure } (+) \odot \text{recip } x \odot \text{sumrecips } xs$$

In this definition, we recognise the application of recip to each element in the list, together with the application of the pure list algebra (+, 0) lifted to the applicative functor. This list algebra is the argument one would provide to foldr, in the definition of the sum function for a list of pure values. Alternatively, we now consider a traversal over a list with the recip applicative action using the instance of traverse for [], defined in Example 4. Such a function, when applied to a list of numbers, results in a list of the reciprocals lifted to Maybe, or in Nothing if there exists any 0 value in the list.

$$\text{recips} :: [\text{Float}] \rightarrow \text{Maybe } [\text{Float}]$$

$$\text{recips } = \text{traverse } \text{recip}$$

332
After a quick inspection of the definitions of \( \text{traverse} \) in Example 4, taking \( i = \text{recip} \), and \( \text{sumrecips} \) we observe that they share the same structure, but in the latter the initial algebra \((:), []\) is replaced by \((+), 0\) avoiding the construction of the reciprocals’ list. We can express \( \text{sumrecips} \) as the result of the applicative combination of a pure fold with a traversal, i.e.:

\[
\text{sumrecips} = \text{fmap} \ (\text{foldr} \ (+) \ 0) \circ \text{traverse} \ \text{recip}
\]

This definition reinforces our observation above: the role of \( \oplus \) and \( \text{pure} \) in the recursive definition is twofold, the mapping of a pure \text{fold} through the applicative function and the effectful traversal with an applicative action.

We consider now one of McBride and Paterson’s original motivating examples [16], the one and only which is not simply a \text{traversal}: an evaluator of arithmetical expressions in the \text{environment} monad.

**Example 6 (Evaluation of Arithmetical Expressions).** Consider the following inductive datatype implementing arithmetical expressions with variables and the datatype of environments consisting of a list of bindings.

\[
\text{data} \ \text{Exp} \ a = \text{Var} \ a \mid \text{Val} \ \text{Int} \mid (\text{Exp} \ a) \oplus (\text{Exp} \ a)
\]

\[
\text{type} \ \text{Env} \ a = [(a, \text{Int})]
\]

The \((\rightarrow) (\text{Env} \ a)\) datatype is an applicative functor.

\[
\text{instance} \ \text{Applicative} \ ((\rightarrow) (\text{Env} \ a)) \text{ where}
\]

\[
\begin{align*}
\text{pure} \ x &= \lambda_\rightarrow \ x \\
\text{ef} \oplus \text{es} &= \lambda_\rightarrow (\text{ef} \ e) \ (\text{es} \ e)
\end{align*}
\]

An evaluator for this language can be expressed applicatively, given a function \( \text{fetch} :: a \rightarrow \text{Env} \ a \rightarrow \text{Int} \) which looks up the value in the environment:

\[
\begin{align*}
\text{eval} &:: (\text{Eq} \ a) \Rightarrow \text{Exp} \ a \rightarrow \text{Env} \ a \rightarrow \text{Int} \\
\text{eval} \ (\text{Var} \ x) &= \text{fetch} \ x \\
\text{eval} \ (\text{Val} \ i) &= \text{pure} \ i \\
\text{eval} \ (e1 \oplus e2) &= \text{pure} \ (+) \oplus \text{eval} \ e1 \oplus \text{eval} \ e2 \\
\text{fetch} &:: (\text{Eq} \ a) \Rightarrow a \rightarrow \text{Env} \ a \rightarrow \text{Int} \\
\text{fetch} \ s \ ((t, \ x) : \text{xs}) &= \text{if} \ (s \equiv t) \text{ then } x \text{ else } \text{fetch} \ s \ \text{xs}
\end{align*}
\]

We want to show that this function has an equivalent definition which satisfies the “pure fold mapped after traversal” pattern recognised in the previous example. First, it should be noticed that \( \text{fetch} \) is an applicative action which can be an argument of the instance of \text{traverse} for \text{Exp} \ a. This traversal will generate an expression of type \text{Exp Int} lifted into the \text{Env Int} \rightarrow \_ \text{ applicative functor, i.e. an arithmetical expression where variables have been replaced by values obtained from the environment.}
instance Traversable Exp where
  traverse ι (Var xs) = pure Var ⊛ ι xs
  traverse ι (Val n) = pure (Val n)
  traverse ι (e1 ⊕ e2) = pure (⊕) ⊛ traverse ι e1 ⊛ traverse ι e2

filledExp :: Eq a ⇒ Exp a → Env a → Exp Int
filledExp = traverse fetch

Thus, an evaluator for pure Exp Int expressions can be defined in terms of the fold for the Exp a datatype:

foldExp :: (a → b) → (Int → b) → (b → b → b) → Exp a → b
foldExp η ζ φ (Var x) = η x
foldExp η ζ φ (Val n) = ζ n
foldExp η ζ φ (e1 ⊕ e2) = foldExp η ζ φ e1 '⊕' foldExp η ζ φ e2
eval' :: Exp Int → Int
eval' = foldExp id id (+)

Finally, mapping eval' after filledExp results in an implementation that reflects the desired pattern:

eval :: Eq a ⇒ Exp a → Env a → Int
eval = fmap (foldExp id id (+)) ◦ traverse fetch

In the previous examples we recognised a common pattern: applicative recursive functions can be understood as the result of mapping a pure fold after a traversal. We propose this pattern as a specification of structural recursion with applicative effects. We will call this operator ifold, idiomatic or applicative fold, and we will show that this operator can be derived from the pure fold mapped after a traversal pattern. Given a bitraversable bifunctor s, an algebra φ :: s b x → x for the functor (s b) and an applicative action ι :: a → f b for an applicative functor f, we specify the ifold operator by the following equation:

ifold φ ι = fold (fmap φ ◦ bitraverse ι id)

This characterisation of ifold is perhaps, too inefficient, as an intermediate data structure is built by traverse with no other purpose than being immediately consumed by the mapping of the fold. However, we can eliminate the intermediate structure using generalised fold-fold fusion (Law 2) and obtain:

fmap (fold φ) ◦ traverse ι = fold (fmap φ ◦ bitraverse ι id)  (2)

This equation motivates the following more efficient datatype-generic definition which can be instantiated to any bitraversable datatype

ifold :: (Applicative f, Bitraversable s) ⇒
  (s b x → x) → (a → f b) → µ (s a) → f x
ifold φ ι = fold (fmap φ ◦ bitraverse ι id)
Example 7 (Lists). An specialised definition of ifold for polymorphic lists, \([a]\), can be derived from the generic definition above.

\[
ifold \:: (Applicative\ f) \Rightarrow (b \rightarrow x \rightarrow x) \rightarrow (x \rightarrow (a \rightarrow f\ b) \rightarrow [a] \rightarrow f\ x
\]

\[
ifold\ \phi\ \eta\ \iota\ [ ] = \text{pure}\ \eta
\]

\[
ifold\ \phi\ \eta\ \iota\ (x : xs) = \text{pure}\ \phi \otimes \iota\ x \otimes \ifold\ \phi\ \eta\ \iota\ xs
\]

The \textit{sumrecips} function defined in Example 5 can thus be expressed in terms of ifold, resulting in the following definition:

\[
sumrecips :: [\text{Float}] \rightarrow \text{Maybe\ Float}
\]

\[
sumrecips = \ifold\ (+)\ 0\ \text{recip}
\]

Example 8 (Expressions). The ifold for the \textit{Exp\ a} datatype introduced in Example 6 is:

\[
ifold \:: (Applicative\ f) \Rightarrow (\text{Int} \rightarrow x) \rightarrow (b \rightarrow x) \rightarrow (x \rightarrow x \rightarrow x) \rightarrow (a \rightarrow f\ b) \rightarrow \text{Exp}\ a \rightarrow f\ x
\]

\[
ifold\ \phi\ \zeta\ \gamma\ \iota\ \text{(Val}\ i) = \text{pure}\ \phi\ i
\]

\[
ifold\ \phi\ \zeta\ \gamma\ \iota\ \text{(Var}\ x) = \text{pure}\ \zeta \otimes \iota\ x
\]

\[
ifold\ \phi\ \zeta\ \gamma\ \iota\ \text{(p \oplus\ q)} = \text{pure}\ \gamma \otimes \ifold\ \phi\ \zeta\ \gamma\ \iota\ p \otimes \ifold\ \phi\ \zeta\ \gamma\ \iota\ q
\]

Then, the evaluator for \textit{Exp\ a} arithmetical expressions, \textit{eval} can be written as:

\[
evalExp :: \text{Eq\ a} \Rightarrow \text{Exp}\ a \rightarrow \text{Env}\ a \rightarrow \text{Int}
\]

\[
evalExp = \ifold\ \text{id}\ \text{id}\ (+)\ \text{fetch}
\]

The ifold operator captures elegantly applicative computations, providing a clear separation of pure algebras and computational effects.

4.2 Applicative build

Dually to what we did in the previous section, we define an applicative build to be an ordinary build followed by a traversal. Since the datatype-generic \textit{traverse} is a \textit{fold} we can use shortcut fusion to obtain:

\[
traverse\ \iota\ \circ\ \build\ \text{g} = \text{g (fmap}\ \text{In} \circ\ \bitraverse\ \iota\ \text{id})
\]

This equation motivates the following definition

\[
ibuild :: (Applicative\ f,\ Bitraversable\ s) \Rightarrow (a \rightarrow f\ d) \rightarrow (\forall b.(s\ a\ b \rightarrow b) \rightarrow c \rightarrow b) \rightarrow c \rightarrow f\ (\mu\ (s\ d))
\]

\[
ibuild\ \iota\ \text{g} = \text{g (fmap}\ \text{In} \circ\ \bitraverse\ \iota\ \text{id})
\]
4.3 Structured Shortcut Fusion

We finally reach to the point where we can present our shortcut fusion rules for applicative computations. Traversal of the intermediate structure proves to be of paramount importance. The following theorem considers the different manners in which we can view the fusion of intermediate structures with applicative effects introduced by a traversal.

**Theorem 1 (Applicative Shortcut Fusion).** Let $\phi :: s \cdot y \cdot b \to b$ be an algebra for the bifunctor $s$, $\iota :: x \to f \cdot y$ an action for the applicative functor $f$, and $g :: (\forall z. (s \cdot x \cdot z \to z) \to c \to z$. Then the following functions of type $c \to f \cdot b$ are equal:

1. $fmap (fold \phi) \circ traverse \iota \circ build g$
2. $ifold \phi \iota \circ build g$
3. $fmap (fold \phi) \circ ibuild \iota \circ g$
4. $g (fmap \phi \circ bitraverse \iota \circ id)$

**Proof.** $(1) = (2)$ by specification of ifold. By equation 2 and shortcut fusion $(2) = (4)$. Finally by equation 3 $(1) = (3)$

We can see applicative shortcut fusion in different ways, with the effects being generated by an $ibuild$, when constructing the intermediate structure, or by an $ifold$, when consuming it. However, in all the cases it is a traversal that is actually generating the effects. Because the intermediate structure is inductive, traversability over the whole structure translates to (bi)traversability of its signature functor in the fused expression.

**Example 9.** We will show an application of Theorem 1 for lists and the $Maybe$ applicative functor. Consider the following function:

$\text{diffList} :: [\text{Float}] \to [\text{Float}] \to [\text{Float}]$

$\text{diffList} \; ys \; [] = []$
$\text{diffList} \; [] \; (x : xs) = []$
$\text{diffList} \; (y : ys) \; (x : xs) = (y - x) : \text{diffList} \; ys \; xs$

diffList $ys \; xs$ calculates the list of differences between values in $ys$ and $xs$, synching the size of the lists. This function is a *good producer*, and thus can be expressed in terms of the $build$ operator for lists:

$\text{build} :: (\forall b. (a \to b \to b) \to b \to c \to b) \to c \to [a]$
$\text{build} \; g = g \; () \; []$
$\text{gendifflist} :: [\text{Float}] \to (\text{Float} \to \text{Float} \to \text{Float}) \to \text{Float} \to [\text{Float}] \to \text{Float}$
$\text{gendifflist} \; ys \; c \; n \; [] = n$
$\text{gendifflist} \; [] \; c \; n \; (x : xs) = n$
$\text{gendifflist} \; (y : ys) \; c \; n \; (x : xs) = (y - x) \; 'c' \; (\text{gendifflist} \; ys \; c \; n \; xs)$
$\text{diffList} :: [\text{Float}] \to [\text{Float}] \to [\text{Float}]$
$\text{diffList} \; ys = \text{build} \; (\text{gendifflist} \; ys)$
We want to define a function \( \text{sumRecipDiffs} \ ys = \text{sumrecips} \circ \text{diffList} \ ys \), where \( \text{sumrecips} \) is the function in Example 5. The function \( \text{sumRecipDiffs} \) verifies the \( \text{fmap} \ (\text{fold} \ \phi) \circ \text{traverse} \ \iota \circ \text{build} \ g \) pattern of Theorem 1, which for lists yields:

\[
\text{map} \ (\text{foldr} \ \phi \ \eta) \circ \text{traverse} \ \iota \circ \text{build} \ g \\
= g \, (\lambda x \ csxs \rightarrow \text{pure} \ \phi \odot \iota \ x \odot csxs) \, (\text{pure} \ \eta)
\]

Thus we can calculate an efficient definition for \( \text{sumRecipDiffs} \ ys \)

\[
\text{sumRecipDiffs} \ ys \\
\equiv \{ \text{sumRecipDiffs specification} \} \\
\text{sumrecips} \circ \text{diffList} \ ys \\
\equiv \{ \text{definitions} \} \\
\text{fmap} \ (\text{foldr} \ (+) \ 0) \circ \text{traverse} \ \text{recip} \circ \text{build} \ (\text{gendifflist} \ ys) \\
\equiv \{ \text{Applicative Shortcut Fusion for Lists (4)} \} \\
\text{gendifflist} \ ys \, (\text{fun} x \ csxs \rightarrow \text{pure} \ (+) \odot \text{recip} \ y \odot csxs) \, (\text{pure} \ 0)
\]

Finally, unfolding the definition of \( \text{gendifflist} \), we get a monolithic definition for \( \text{sumRecipDiffs} \) which avoids the construction of the two intermediate lists:

\[
\text{sumRecipDiffs} \ :: \ [\text{Float}] \rightarrow [\text{Float}] \rightarrow \text{Maybe Float} \\
\text{sumRecipDiffs} \ ys \ [] = \text{pure} \ 0 \\
\text{sumRecipDiffs} \ [] \ xs = \text{pure} \ 0 \\
\text{sumRecipDiffs} \ (y : ys) \ (x : xs) = \text{pure} \ (+) \odot \text{recip} \ (y - x) \\
\odot \text{sumRecipDiffs} \ ys \ xs
\]

5 Conclusions and Future Work

We have presented a shortcut fusion rule for applicative computations. While one can apply the generalised shortcut fusion rule to applicative computations, the rule does not capture their structure. We aimed at obtaining a more structured fusion law that took into account the way applicative computations are written. By analysing several examples we found that traversals are at the core of applicative computations, and hence we proposed the pattern \( \text{build-traverse-fold} \) as the core of structural applicative computations and introduced a shortcut fusion result for those patterns. Fusing ordinary folds and builds with traversals yields the \( \text{ifold} \) and \( \text{ibuild} \) operators, which elegantly separate the pure part of the computation from the one producing computational effects.

Future Work The proposed pattern arose as a result of the study of several examples found in the literature. Despite the elegance of the results, we would like to obtain a more theoretically founded justification for them such as an
initial algebra semantics for ifold. Related to this is the notion of a category of applicative computations, but this notion is still missing.

In order to reason about traversals, coherence laws for traverse are needed. Gibbons and Oliveira [10] propose some very reasonable laws for traverse, but we think that they should arise directly from the structure. Consequently, further research is needed on the foundations of applicative functors.

References

Abstract. We propose the use of dependent types for functional programming in a combinator-style fashion, to replace complex proofs of programs by a construction from building blocks which provide parts of the proof. Three examples with increasing level of sophistication demonstrate our approach: sized powerlists used by the Fast Fourier Transform, a purely functional queue with amortised cost linear in the number of elements processed and guarantees on the number of steps required by an autonomous vehicle to move from one location to another. We present our example programs in the language Agda, a functional programming language for programming with full-spectrum dependent types.

1 Introduction

Dependently typed programming, in which functions are predicated on types to give stronger static correctness guarantees, has been an emerging trend in functional programming for a number of years. Recent developments include languages such as Agda (10) and Idris (1) which allow larger scale programming with dependent types with applications in security (9), domain-specific languages (DSLs) (12), parsers (3) and compilers (7). However, if this trend is to continue, we need to ensure that program verification is not achieved at the expense of programmer productivity. In particular it is crucial to avoid the need for reasoning and providing too many details of proofs, but rather allow a programmer to focus on the domain of interest.

In this paper we describe the value of combinator-style programming with dependent types. Rather than developing post-hoc proofs of program correctness, we implement small operations with a clearly defined meaning expressed as a dependent type. We show how combining such operations allows the development of increasingly complex systems, avoiding the need for complex correctness proofs. We will gradually introduce power of dependently typed programming. In the next section we introduce a lightweight form by having static size information for vectors and applying this to structure the Fast Fourier Transform algorithm. In Section 3 we use dependent types to verify that the implementation of a purely functional queue works with a number of operations linear in
the number of enqueued elements, using the amortisation principle. In Section 4 we build a more complex example, providing static guarantees for autonomous vehicle movements based on an embedded DSL. Note that the guarantees refer to the abstract properties of the implementation of the queue or vehicle modelled in Agda, not necessarily to operational properties when interpreting the program in the Agda run-time environment. Section 5 discusses related work and Section 6 concludes.

2 Vectors, Powerlists and the Fast Fourier Transform

In the Agda standard library there is a data type of vectors defined which, in addition to the polymorphic element type (here \(a\)), carry an index which describes the length of the vector. These types are similar to GADTs in Haskell and have originally been named \textit{indexed families}.

\begin{verbatim}
infixr 5 ::-

data Vec {a} (A : Set a) : N → Set a where
  [ ] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

Agda uses a mixfix notation in which operators are defined by stating the positions of arguments by an underscore, as in \(\_::\_\) which conses an element to a vector. Curly braces denote implicit arguments which are mainly used to introduce variable names for values. They can optionally be instantiated at the application context, in particular to provide necessary information to the type checker. Note that while \(a\) scopes over the entire definition of \(\text{Vec}\), indices such as \(n\) are only valid for one constructor definition or, in the context of functions, in a single defining equation.

On these vectors the append operator can be defined quite similar to the definition as we would know from Haskell or ML. Note that we need to update the size information in the type index appropriately and Agda will check whether this complies with the implementation.

\begin{verbatim}
infixr 5 _+_+

_+_+ : ∀ {a m n} {A : Set a} → Vec A m → Vec A n → Vec A (m +^N n)
[ ] +^ y = y
(x :: xs) +^ y = x :: (xs +^ y)
\end{verbatim}

Special kinds of Divide-and-Conquer algorithms require to split a vector in two parts of the same size to apply vector operations on both parts. For this purpose, one can use a special kind of vector named Powerlist(8). A formulation of a Powerlist in Agda could be as follows:

\begin{verbatim}
data PList (a : Set) : N → Set where
  ⟨_⟩ : a → PList a 0
  _|_ : {n : N} → PList a n → PList a n → PList a (suc n)
\end{verbatim}
Since Powerlists are always of a size a power of 2, we use the logarithm of their length by an index. Of course, we want to convert vectors (if their length is a power of 2) into powerlists and vice versa.

We need a function calculating the powers of 2 and a law that adding zero to the right is the identity:

\[
\begin{align*}
pow2 & : \mathbb{N} \rightarrow \mathbb{N} \\
pow2 \ 0 & = \ 1 \\
pow2 \ (\text{suc} \ n) & = \ 2 \ \ast^{\mathbb{N}} \ pow2 \ n
\end{align*}
\]

\[
\begin{align*}
\text{Neut} + 0 & : \{ \ n : \mathbb{N} \} \ \rightarrow \ (n +^{\mathbb{N}} \ 0 \ \equiv \ n)
\end{align*}
\]

This law can be proved within Agda by induction, specifying a case for 0 and \text{suc} \ n:

\[
\begin{align*}
\text{Neut} + 0 \ \{0\} & = \ \text{refl} \\
\text{Neut} + 0 \ \{\text{suc} \ n\} & = \\
& \begin{cases} 
\text{(suc} \ n) +^{\mathbb{N}} \ 0 & \text{-- applying evaluation rule for +} \\
\text{suc} (n +^{\mathbb{N}} \ 0) & \text{-- applying induction hypothesis} \\
\text{suc} \ n
\end{cases}
\end{align*}
\]

If we would like to omit the proof (e.g., because it is provided by an external tool), we can just write the word postulate in front of the law. We will do this in the rest of the paper where proofs are just simple arithmetic and could be proven automatically using the ring solver from the Agda standard library.

We need the Law \text{Neut} + 0 to rewrite the type of the right-hand side of the defining equation to match the syntactic structure in the type signature.

\[
\begin{align*}
\text{convPV} & : \{ \ m : \mathbb{N} \} \ \{ \ a : \text{Set} \} \ \rightarrow \ \text{PList} \ a \ m \ \rightarrow \ \text{Vec} \ a (\ pow2 \ m) \\
\text{convPV} \ < \ a > & = \ a :: [] \\
\text{convPV} \ \{ \text{suc} \ m\} \ (h | l) \ \text{rewrite} \ \text{Neut} + 0 \ \{ \pow2 \ m\} \\
& = \ \text{convPV} \ \{m\} \ h + \ \text{convPV} \ \{m\} \ l
\end{align*}
\]

Likewise, we can convert a list of size a power of 2 into a powerlist using the \text{splitAt} function from the Agda library. The index \text{pow2} \ m for the function argument already enforces the argument to be of the right size.

\[
\begin{align*}
\text{convPV} & : \{ \ m : \mathbb{N} \} \ \{ \ a : \text{Set} \} \ \rightarrow \ \text{Vec} \ a (\ pow2 \ m) \ \rightarrow \ \text{PList} \ a \ m \\
\text{convPV} \ \{ \text{zero}\} \ (a :: []) & = < a > \\
\text{convPV} \ \{ \text{suc} \ m\} \ xs \ \text{with} \ \text{splitAt} \ (\pow2 \ m) \ xs \\
& \ldots \ | \ (h, l, \cdot) \ \text{rewrite} \ \text{Neut} + 0 \ \{ \pow2 \ m\} = \ \text{convPV} \ \{m\} \ h + \ \text{convPV} \ \{m\} \ l
\end{align*}
\]

Similar to map and zipWith in Haskell and the Agda library for vectors, we define data-parallel operations on one or two Powerlists:
mapP : \( \{ a, b : \text{Set} \} \{ m : \mathbb{N} \} \rightarrow (a \rightarrow b) \rightarrow \text{PList} a m \rightarrow \text{PList} b m \)
mapP \( f < a > = < f a > \)
mapP \( f (ah \mid al) = (\text{mapP} f ah \mid \text{mapP} f al) \)
zipWithP : \( \{ a, b, c : \text{Set} \} \{ m : \mathbb{N} \} \rightarrow (a \rightarrow b \rightarrow c) \rightarrow \text{PList} a m \rightarrow \text{PList} b m \rightarrow \text{PList} c m \)
zipWithP \( f < a > < b > = < f a b > \)
zipWithP \( f (ah \mid al) (bh \mid bl) = (\text{zipWithP} f ah bh) \mid (\text{zipWithP} f al bl) \)

**Fig. 1.** Recursive structure of FFT: even/odd decompose, left/right compose

The elegance of Misra’s powerlist formalism (8) is that it allows to decompose a non-singleton powerlist by pattern matching not only into left and right part but also into subsequences with odd and even indices, as is useful for the Fast Fourier Transform (see Figure 1) and several other algorithms with a butterfly dependence structure.

In order to pattern match in a real programming language in a different way than the construction has happened, Wadler introduced the principle of views (14). In a similar way we use an alternative data structure for pattern matching against the subsequences with odd and even indices. The conversion requires an unshuffle operation to preserve the semantics:

**data** \( \text{PList}^{\triangledown} (a : \text{Set}) : \mathbb{N} \rightarrow \text{Set} \) **where**

\[ \triangledown : \{ m : \mathbb{N} \} \rightarrow \text{PList} a m \rightarrow \text{PList} a m \rightarrow \text{PList}^{\triangledown} a (\text{suc} m) \]

\[ \text{unshuffle} : \{ m : \mathbb{N} \} \{ a : \text{Set} \} \rightarrow \text{PList} a (\text{suc} m) \rightarrow (\text{PList} a m \times \text{PList} a m) \]

\[ \text{unshuffle} \{ \text{zero} \} (< a > \mid < b >) = (< a >, < b >) \]
unshuffle \( \{ \text{suc } m \} \) (\( xs \mid ys \)) with unshuffle \( \{ m \} \) \( xs \mid \) unshuffle \( \{ m \} \) \( ys \)

\[
\begin{align*}
&... \mid (xe, xo) \mid (ye, yo) = ((xe \mid ye), (xo \mid yo))
\end{align*}
\]

Function \textit{decompose} then converts the top level of a powerlist into the new data type. Note that this definition is denotational and leaves open the possibility for a more efficient (and parallel) implementation than by the unshuffle function.

\[
\begin{align*}
&\text{decompose}^\triangleleft \colon \{ m : \mathbb{N} \} \{ a : \text{Set} \} \rightarrow \text{PList} \ a (\text{suc } m) \rightarrow \text{PList}^\triangleright \ a (\text{suc } m) \\
&\text{decompose}^\triangleleft \ x s \text{ with unshuffle } x s
\end{align*}
\]

Instead of approximating complex numbers we use \( \mathbb{Z}/p \) as the vector element domain. For a prime number \( p \) we then have \( p-1 \) roots of unity available.

The data type \textit{Fin }\( p \) denotes the natural numbers from 0 to \( p-1 \), and the remainder operation \(( \mod \) for \(+/-, \times)\) e.g.:

\[
\begin{align*}
&+^p : (p : \mathbb{N}) \rightarrow \text{Fin } p \rightarrow \text{Fin } p \\
&+^0 (\text{suc } p) \ x \ y = (\text{toN } x +^\mathbb{N} \text{toN } y) \mod (\text{suc } p)
\end{align*}
\]

Vectorised arithmetic operations (\( \boxplus/\boxminus/\boxtimes \) for \(+/-, \times)\), e.g.:

\[
\begin{align*}
&\oplus - : \{ m p : \mathbb{N} \} \rightarrow \text{PList} (\text{Fin } p) \ m \rightarrow \text{PList} (\text{Fin } p) \ m \rightarrow \text{PList} (\text{Fin } p) \ m \\
&\oplus - \{ m \} \{ p \} \ a \ b = \text{zipWithP} (+^p \ p) \ a \ b
\end{align*}
\]

Function \textit{powers} computes the roots of unity required for the FFT algorithm:

\[
\begin{align*}
&\text{powers} : \{ p : \mathbb{N} \} \{ \cdot : \text{Fin } p \} (m : \mathbb{N}) \rightarrow \text{PList} (\text{Fin } p) \ m \\
&\text{powers} \{ 0 \} \{ () \} \cdot \\
&\text{powers} \{ \text{suc } p \} \{ q \} \ 0 \text{ with } p \\
&... \mid 0 = < \# 0 > \\
&... \mid \text{suc } \cdot = < \# 1 > \\
&\text{powers} \{ \text{suc } p \} \{ q \} \ (\text{suc } n) \text{ with powers} \{ \text{suc } p \} \{ q \} \ n \\
&... \mid xs = \text{shuffle} \ x s \ (\text{mapP} (x \rightarrow \ast^p \ (\text{suc } p) (\text{root } (\text{suc } p) \{ q \} \ n) \ x) \ x s)
\end{align*}
\]

Finally, we can state the FFT algorithm close to Misra’s form (8):

\[
\begin{align*}
&\text{fft} : \{ p : \mathbb{N} \} \{ \cdot : \text{Fin } p \} (m : \mathbb{N}) \rightarrow \text{PList} (\text{Fin } p) \ m \\
&\text{fft} \{ 0 \} \{ () \} \cdot \\
&\text{fft} \{ \text{suc } p \} \{ q \} \{ \text{suc } m \} \ x s \text{ with } \text{decompose}^\triangleleft \{ m \} \ x s
\end{align*}
\]

\[
\begin{align*}
&... \mid (l \# r) = (L \boxplus uR) \mid (L \boxminus uR) \text{ where} \\
&L = \text{fft} \{ \text{suc } p \} \{ q \} \{ m \} \ l \\
&R = \text{fft} \{ \text{suc } p \} \{ q \} \{ m \} \ r \\
uR = R \boxtimes \text{powers} \{ \text{suc } p \} \{ q \} \ m
\end{align*}
\]
3 Purely Functional Queue with Amortised Linear Cost

We can implement a queue in a purely functional language that operates in time linear with the number of elements processed (11). This has been of particular interest for us in the analysis of worst-case execution time (6) of programs in the language Hume (4).

The implementation uses two stacks (Vec), A and B, depicted in Figure 2. If an element is to be dequeued (Figure 2(a)), it is taken from the top of Stack B if Stack B is not empty. Otherwise Stack A is reversed (Figure 2(b),(c)) and becomes the new Stack B while the new Stack A becomes empty. Elements to be enqueued are just pushed onto Stack A (Figure 2(d)).

So if an element is to be dequeued with an empty Stack B, it triggers a number of operations proportional to the size of A, say n. However, this reversal allows it in the future to dequeue n–1 elements without the need for a reversal. One says that the overhead for reversals is amortised among all elements of the queue processed.

We like to define an implementation of the queue in Agda that comes with a certificate that for n elements inserted into the queue at most 4*n simple stack operations (test for emptiness not included) are required.

In order to create a data type for the queue with two stacks A and B we are suggesting an invariant to express the amortisation in terms of four parameters and how these parameters are modified by each push/pop operation on Stack A/B. We implement the amortisation by making each enqueuing request paying as much tokens as operations are required to process the element, i.e., four. The variables are a and b for the number of elements on Stack A and B, p the amount of tokens paid and c the operations consumed so far. The invariant is then:

\[
\text{invQ} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set}
\]

\[
\text{invQ } a \ b \ p \ c = (p \equiv c +^N 3 *^N a +^N b)
\]

The amount of tokens p we have paid equals the cost c for the elements that have already left the queue plus the potential of the elements in the queue. Each
element on Stack B carries a potential of 1 which is consumed to dequeue it by a
simple pop. Each element on Stack A carries a potential of 3, because it requires
two more operations to move it onto Stack B, namely a pop from A and a push
onto B.

Let us now consider how each operation will modify the variable instantiation
while preserving the invariant, of course: Each of the push/pop operations will
increase the consumed resource count by one. A push on A increases the paid
tokens counter p by four units. Each push/pop will increment/decrement the
number of elements on the involved stack by one. Furthermore, a pop from
Stack A will return us two tokens which the internal implementation can then
use to pay for the push onto Stack B.

We define the language of all possible queue operation sequences by the data
type \textit{Queue} which carries an invariant based on the four parameters \(a, b, p,\) and\(c.\) However, we have find the right balance which parameters we expose to the
outside. If we hide all of them, we will not be able to expose the amount of tokens
to be paid to the user of the data type. On the other hand, if we also expose \(a\)
and \(b,\) then the user will have to deal with the internal state of the stack which
will severely break the modularity of our data type implementation. The same
holds for \(c\) because the user is not interested in the fact how much elements have
been processed so far, which depend on the access history. Therefore, we only
make the amount of tokens paid so far visible, and the element type, of course.
The other parameters and the proof of the invariant become fields of the data
type and are updated by its use. The invariant provides us with a guarantee that
after all elements have been processed \((a = 0 \text{ and } b = 0)\) we have paid as much
tokens as operations done and this was four times the number of elements, since
we paid four tokens to be able to enqueue each element.

\begin{verbatim}
record Queue (elem : Set) (paid : N) : Set where
  constructor mkQ
  field
    a b c : N
    inv : paid ≡ c + N 3 * N a + N b
    sA : Vec elem a
    sB : Vec elem b

The initialisation of the queue:

initQueue : { elem : Set } → Queue elem 0
initQueue = mkQ 0 0 0 refl [] []

Enqueuing: we need to rewrite the invariant to fit the new values of \(a\) and \(c\)
in combination with the increase of payment by four units.

postulate modPushA : { a b p c : N } → invQ a b p c
  → invQ (suc a) b (p + N 4) (suc c)
enqueue : { paid : N } { elem : Set } → elem → Queue elem paid
\end{verbatim}
\[ \text{enqueue} \{ p \} x (\text{mkQ} \ a \ b \ c \ \text{inv} \ sA \ sB) = \text{mkQ} (\text{suc} \ a) \ b (\text{suc} \ c) (\text{modPushA} \ \{ a \} \ \{ b \} \ \{ p \} \ \{ c \} \ \text{inv}) (x :: sA) \ sB \]

For dequeuing, we define an auxiliary function \text{reverseS} which moves the elements from Stack A to Stack B. Each time we move an element we have to rewrite the invariant.

The \text{dequeue} operation returns an element of a \text{Maybe} type, i.e., \text{nothing} if the queue is empty, otherwise just with the first element of the queue.

\begin{align*}
\text{postulate ModRevS} : \{ a \ b \ p \ c : \mathbb{N} \} & \rightarrow (p \equiv c + \mathbb{N} 3 \ast ^N (\text{suc} \ a) + ^N b) \\
\text{postulate ModPopB} : \{ a \ b \ p \ c : \mathbb{N} \} & \rightarrow \text{invQ} a (\text{suc} \ b) \ p \ c
\end{align*}

\text{reverseS} : \{ \text{paid} : \mathbb{N} \} \{ \text{elem} : \text{Set} \} \rightarrow \text{Queue elem paid} \rightarrow \text{Queue elem paid}
\text{reverseS} (\text{mkQ} 0 \ b \ c \ \text{inv} \ sA \ sB) = \text{mkQ} 0 \ b \ c \ \text{inv} \ sA \ sB
\text{reverseS} \{ p \} (\text{mkQ} (\text{suc} \ a) \ b \ c \ \text{inv} (x :: sA) \ sB)
= \text{reverseS} (\text{mkQ} a (\text{suc} \ b) (\text{suc} \ c))
(\text{ModRevS} \ \{ a \} \ \{ b \} \ \{ p \} \ \{ c \} \ \text{inv}) \ sA (x :: sB))
\text{dequeue} : \{ \text{paid} : \mathbb{N} \} \{ \text{elem} : \text{Set} \}
\rightarrow \text{Queue elem paid} \rightarrow \text{Maybe} (\text{Queue elem paid} \times \text{elem})
\text{dequeue} (\text{mkQ} 0 \ 0 \ \cdots \ \cdots) = \text{nothing}
\text{dequeue} \{ p \} (\text{mkQ} a (\text{suc} \ b) \ c \ \text{inv} \ sA (x :: sB))
= \text{just} (\text{mkQ} a \ b (\text{suc} \ c) (\text{ModPopB} \ \{ a \} \ \{ b \} \ \{ p \} \ \{ c \} \ \text{inv}) \ sA \ sB, x)
\text{dequeue} \{ p \} (\text{mkQ} (\text{suc} \ a) \ 0 \ c \ \text{inv} \ sa \ sb)
\text{with} \text{reverseS} (\text{mkQ} (\text{suc} \ a) \ 0 \ c \ \text{inv} \ sa \ sb)
\ldots | (\text{mkQ} a1 \ 0 \ \cdots \ \cdots) = \text{nothing}
\ldots | (\text{mkQ} a1 (\text{suc} \ b1) \ c1 \ \text{inv1} \ sA1 (x :: sB1))
= \text{just} (\text{mkQ} a1 \ b1 (\text{suc} \ c1))
(\text{ModPopB} \ \{ a1 \} \ \{ b1 \} \ \{ p \} \ \{ c1 \} \ \text{inv1}) \ sA1 \ sB1, x)

4 Autonomous Vehicle Operations

When operating autonomous vehicle in resource-critical missions there is the need to verify resource properties such as energy consumption or time to complete a mission (5). If parameters are involved as well as sophisticated formulae (especially beyond polynomial expressions) automatic analysis or verification might fail, due to the limits of computability. In this case, dependently typed programs can be especially helpful because the programmer does (or should) have the knowledge about how the system will behave. The difficulty that remains is how to express this as a certificate in the program.

4.1 Verifying Properties of AV Actions

Let us assume a scenario in which autonomous vehicles can be located at the points of an infinite two-dimensional integer grid and can be faced to one of the
directions North (N), South (S), West (W) or East (E). Furthermore they are supposed to have the following operations:

- Initialisation: the vehicle is dropped at a particular location with particular orientation and initialised with the global time.
- Step: the vehicle can move to the next grid point in the current orientation. This will cost one unit of time.
- Turn: the vehicle can change orientation by turning left, i.e. (N → W, W → S, S → E, E → N). A single turn operation takes one unit of time. We can show that each direction can be reached with a maximum of three single turn operations.
- Wait: the vehicle can wait for a given number of time units without performing any action. In a simple model this serves for a static synchronisation of time. In more advanced models involving power as a resource, the AV could recharge its batteries from solar power while waiting.

Figure 3 shows a sequence of actions such an AV can perform if it is oriented towards North and wants to reach a location which is 6 units of length West and 4 units of length North. Although there is a shorter sequence, for simplicity we have chosen always to have a single turn left at the intermediate position. \texttt{turnTo} and \texttt{stepI} will be introduced later below. Our aim is that if given a start location \((x_1, y_1)\) and a final location \((x_2, y_2)\) we can generate a sequence of control instructions that will make the AV move from start to finish within \(4 + |x_2 - x_1| + |y_2 - y_1|\) units of time, if each single turn and step takes one unit of time.

4.2 Basic Definitions

The directions of an AV, its time and coordinates are collected in a record of type \texttt{AV}. We will make use of this at the type level, permitting us to defined static
constraints in terms of the AVs components. The constructor \texttt{mkAV} permits access to the record fields by pattern matching.

\begin{verbatim}
data Direction : Set where
    N W S E : Direction

record AV : Set where
    constructor mkAV
    field
        time : N
        dir : Direction
        px : Z
        py : Z

initAV : N → Direction → Z → Z → AV
initAV t d x y = record { time = t; dir = d; px = x; py = y}
\end{verbatim}

4.3 Turning the AV

\begin{verbatim}
turn : Direction → Direction

turn N = W
turn W = S
turn S = E
turn E = N
\end{verbatim}

In order to establish a proof that the vehicle can turn to each direction within 3 simple operations, we provide a mapping from the directions to the set of the first 4 naturals: 0, 1, 2, 3 and a function \texttt{turnN} which carries out a repeated number of turns:

\begin{verbatim}
dirNo : Direction → Fin 4
dirNo N = ♯ 0
dirNo W = ♯ 1
dirNo S = ♯ 2
dirNo E = ♯ 3
turnN : \{ n : N \} → Fin n → Direction → Direction
turnN zero d = d
turnN (suc n) d = turn (turnN n d)
\end{verbatim}

Furthermore, we define a function \texttt{rol} which has the property:

\begin{verbatim}
dirNo (turn n) = rol (dirNo n)
\end{verbatim}

The \texttt{rol} function will subtract one from the number if it is positive and assign it the highest number in the range if it was zero. Note that the reduction from \texttt{suc} \texttt{i} to \texttt{i} would reduce our range and we need to declare its preservation by using \texttt{inject1}.

349
Function \( \Box \) subtracts a number from another modulo 4. It is used to calculate the number of turns required in function \( \text{turnCount} \). Function \( \text{iterate} \) applies a function a given number after each other.

Function \( \text{iterate} \):

\[
\text{iterate} : \{ a : \text{Set} \} (k : \mathbb{N}) \rightarrow (\text{Fin} k \rightarrow (a \rightarrow a)) \rightarrow a \rightarrow a
\]

\[
\text{iterate} (\text{suc} n) f x = \text{iterate} n (i \rightarrow f (\text{suc} i)) (f \# 0) x
\]

Function \( \Box \):

\[
\Box : \{ q : \mathbb{N} \} \rightarrow \text{Fin (suc q)} \rightarrow \text{Fin (suc q)} \rightarrow \text{Fin (suc q)}
\]

Function \( \text{eqDir} \):

\[
\text{eqDir} : \text{Direction} \rightarrow \text{Direction} \rightarrow \text{Bool}
\]

\[
\text{eqDir} \text{N N} = \text{true}
\]

\[
\text{eqDir} \text{W W} = \text{true}
\]

\[
\text{eqDir} \text{S S} = \text{true}
\]

\[
\text{eqDir} \text{E E} = \text{true}
\]

\[
\text{eqDir} \Box = \text{false}
\]

Function \( \text{turnCount} \):

\[
\text{turnCount} : \text{Direction} \rightarrow \text{Direction} \rightarrow \text{Fin} 4
\]

\[
\text{turnCount} \text{from to} = \text{dirNo to} \Box \text{dirNo from}
\]

Now we can deliver a function which returns the number of required turns together with a proof that this number actually establishes that. This is encoded by a dependent pair \( \Sigma (\text{Fin} 4) \ldots \), where the first component, the number is an element of \( \text{Fin} 4 \). This guarantees us that we can establish to turn the AV into each direction with at most three simple turn operations.

Function \( \text{turnProof} \):

\[
\text{turnProof} : \{ \text{from to : Direction} \} \rightarrow \Sigma (\text{Fin} 4) (\lambda c \rightarrow to \equiv \text{turnN c from})
\]

\[
\text{turnProof} \{ \text{N} \} \{ \text{N} \} = (\text{turnCount N N}, \text{refl})
\]

\[
\text{turnProof} \{ \text{N} \} \{ \text{W} \} = (\text{turnCount N W}, \text{refl})
\]

... we omit the other 14 cases

Instead of a complete enumeration of special cases for each combination of from and to we can also use a combinator which generates this enumeration; however this is not much simpler here.

We then lift the \( \text{turn} \) operation to work on an AV, including the increment in time it requires:

Function \( \text{turnAV} \):

\[
\text{turnAV} : \text{AV} \rightarrow \text{AV}
\]

\[
\text{turnAV} (\text{mkAV} t d x y) = \text{mkAV (suc t) (turn d) x y}
\]

### 4.4 Forward movement of the AV

Doing one or several steps forward will change the location depending on the orientation of the AV.
4.5 A language to compose simple AV operations

In the previous section we have defined the state changes on an AV by single and repeated step and move actions. Now, we are using these in a data type for an embedded DSL for AV actions. Programs in this language are constructed in using a bind operator similar as many people will know from monadic programming in Haskell.

The operation in this language are the identity (\texttt{Id}), a wait operation (\texttt{Wait}) which serves to align alternative operations with different durations, a turn (\texttt{Turn}) and step (\texttt{Step}) operation, the bind operator (\texttt{>>=}) and a case distinction depending on the direction (\texttt{IfDir}).

\begin{verbatim}
  nSuc : N → N → N
  nSuc zero x = x
  nSuc (suc n) x = suc (nSuc n x)

  data AVLang : AV → AV → Set where
    Id : { s : AV } → AVLang s s
    Wait : { t : N } { d : Direction } { x y : Z } { k : N}
         → AVLang (mkAV t d x y) (mkAV (nSuc k t) d x y)
    Turn : { s : AV } → AVLang s (turnAV s)
    Step : { s : AV } → AVLang s (stepAV s)
    _>>=_ : { s1 s2 s3 : AV }
         → AVLang s1 s2 → AVLang s2 s3 → AVLang s1 s3
    IfDir : { s1 s2 s3 : AV } { d : Direction } → AVLang s1 s2
         → AVLang s1 s2

  infixl 10 _>>=_
\end{verbatim}

An interpretation function defines the effect programs in this language have on the state of the AV:
interp : \{s1 \ s2 : AV\} \rightarrow AVLang s1 s2 \rightarrow AV \rightarrow AV
interp Id x = x
interp (Wait \cdot) x = x
interp Turn x = turnAV x
interp Step x = stepAV x
interp (r \gg= f) x = interp f (interp r x)
interp (IfDir d x y) s with eqDir (AV.dir s) d
... | true = interp x s
... | false = interp y s

To define repetitions of simple AV operations we are embedding the simple
language into a language with repetitions:

data IterLang : AV \rightarrow AV \rightarrow Set
where
Emb : \{s1 \ s2 : AV\} \rightarrow AVLang s1 s2 \rightarrow IterLang s1 s2
>>>=- : \{s1 \ s2 s3 : AV\} \rightarrow IterLang s1 s2 s3 \rightarrow IterLang s1 s3
StepI : \{s : AV\} (n : N) \rightarrow IterLang s (stepAVI n s)
TurnI : \{s : AV\} (k : N) \rightarrow IterLang s (turnAVI k s)
TurnTo : \{t : N\} \{from : Direction\} (to : Direction) \{x y : Z\}

To implement programs in IterLang, we are using a translation function to
AVLang:

genNSteps : \{s : AV\} (n : N) \rightarrow AVLang s (stepAVI n s)
genNSteps zero = Id
genNSteps (suc n) = genNSteps n >>>= Step
genNTurns : \{n : N\} \{s : AV\} \rightarrow AVLang s (turnAVI n s)
genNTurns zero = Id
genNTurns (suc n) = genNTurns n >>>= Turn
translate1 : \{s1 \ s2 : AV\} \rightarrow IterLang s1 s2 \rightarrow AVLang s1 s2
translate1 (Emb x) = x
translate1 (StepI n) = genNSteps n
translate1 (TurnI n) = genNTurns n

A bit tricky is the translation for turning the AV into a particular direction.
We have presented a proof above (TurnProof) that this operation will always
succeed in at most 3 time steps. However, if it can do in less, the AV would have
to wait to align the times for the different cases. This is an example where we
allow for loss of precision in order to simplify compositionality.

translate1 \{mkAV \cdot from\cdot\} (TurnTo to) with turnProof \{from\} \{to\}
... | (c, proof) with c

352
4.6 A language for complex AV operations

Until now we have execution time guarantees for AV operations repeated in control parameters and based on run-time decisions with a finite case distinction. This section is about guarantees depending on run-time decisions with an unbounded number of cases. As just one example we present a MoveTo operation which moves the AV from any position to any other position and an implementation which comes with a guarantee on the execution time.

\[
\text{data CompLang : AV } \rightarrow \text{ AV } \rightarrow \text{ Set where}
\]
\[
\text{MoveTo : } \{ t1 \ t2 : N \} \{ d1 \ d2 : \text{Direction} \} \{ x1 \ y1 : Z \} \{ x2 \ y2 : Z \}
\rightarrow \text{CompLang (mkAV t1 d1 x1 y1) (mkAV t2 d2 x2 y2)}
\]

We begin with the simple cases in which movements happen only along a single direction:

\[
\text{postulate lawNorth : } \{ m \ t : N \} \{ x \ y : Z \}
\rightarrow (\text{mkAV (nSuc m t) N x (y +Z (+ m))}) \equiv \text{stepAVI m (mkAV t N x y)}
\]
\[
\text{walkNorth : } \{ t : N \} \{ m : N \} \{ x \ y : Z \}
\rightarrow \text{IterLang (mkAV t d x y) (mkAV (nSuc m t) N x (y +Z (+ m)))}
\]

We have similar laws for the other directions.

Then, we compose a turn into this direction with the move, which is the movement into the second direction:

\[
\text{turnWalkNorth : } \{ d : \text{Direction} \} \{ m : N \} \rightarrow \{ t : N \} \{ x \ y : Z \}
\rightarrow \text{IterLang (mkAV t d x y) (mkAV (nSuc m (nSuc 3 t)) N x (y +Z (+ m)))}
\]

\[\text{turnWalkNorth m} = \text{TurnTo N >>=} \text{walkNorth m}\]

The other direction are analogous.

We can then compose the two movements to form a movement between any two points. Again, we show only one of the four possibilities:

\[
\text{postulate LawWestSouth : } \{ t \ m \ n : N \}
\rightarrow (t +N 4 +N m +N n) \equiv \text{nSuc n (suc (nSuc m (nSuc 3 t)))}
\]
\[
\text{walkWestSouth : } \{ d : \text{Direction} \} \{ m \ n : N \} \rightarrow \{ t : N \} \{ x \ y : Z \}
\rightarrow \text{IterLang (mkAV t d x y) (mkAV (t +N 4 +N m +N n))}
\]
\[ S(x - Z(+m)) (y - Z(+n)) \]
\[
\text{walkWestSouth} \{\cdot\} m n \{t\} \text{ rewrite LawWestSouth} \{t\} \{m\} \{n\}
\]
\[
= \text{turnWalkWest} m \gggg \text{Turn} 1 \gggg \text{walkSouth} n
\]

What we can read off from the type (the second component of IterLang) is that the time \( (t) \) has increased by \( 4 + m + n \), where \( m/n \) is the distance in the first/second direction.

Furthermore, we need a case distinction based on the signs of the differences between the final and start coordinates, to calculate the absolute value for the number of movements the vehicle should move in each direction and to apply the appropriate function, e.g. \text{walkWestSouth}.

5 Related Work

From the Nottingham FPLunch weblog we obtain that Wouter Swierstra has done similar work with powerlists in Agda, mentioning the need to have an appropriate view for odd/even decomposition, but we were unable to obtain more details (13). A complete certification of the FFT algorithm in Coq was given by Venanzio Capretta (2). Chris Okasaka has investigated the effect of laziness on the amortisation of purely functional queues (11).

6 Conclusions

We have seen that dependent types can be usefully integrated into functional programs to verify some interesting properties such as correct sizes of lists and powerlists, resource usage amortisation and number of steps required for an autonomous vehicle to move from one point to another. These three examples demonstrate well the complexity one experiences when making the properties more powerful.

We made the experience that it is important to find the right balance which properties to make explicit and which not. There is the danger of breaking modularity of software composition when making the invariant of a data type visible to the outside, as this requires the environment to follow the effect of the internal details.

There is still lots of potential concerning improvements of the examples: the verification of stronger properties for powerlists, e.g. that inverse FFT composed with FFT is the identity, the use of amortised queue as part of a larger program and, for the AV example, the abstraction from concrete directions to reduce the program size as well as the construction of entire plans involving several vehicles, e.g. searching a terrain.

Further potential future work could be the identification of commonalities of combinators between different domains and finding general forms which can be instantiated appropriately. We have already seen that the map/zipWith functions can be easily adapted, but also the Fin data type in combination with the modulo operation turned out to be useful for FFT as well as for the AV example.
Bibliography