## Contents

Preface .......................................................... II

### ICC and Abstract Interpretation (Chair: Marko van Eekelen)


Emanuele Cesena, Marco Pedicini and Luca Roversi. *Typing a Core Binary Field Arithmetic in a Light Logic*. ................................. 8


### Timing Analysis (Chair: Puri Arenas)

Christoph Herrmann and Kevin Hammond. *Resource Analysis with Static Values and Probabilities*. ........................................... 54

Mihail Asavoae and Dorel Lucanu. *Formal Executable Semantics for Timing Analysis*. .......................................................... 64

Nuno Gaspar, Vitor Rodrigues and Simão Melo De Sousa. *Certifying Execution Time*. .......................................................... 79

### Polynomial interpolation (Chair: Hans-Wolfgang Loidl)

Manuel Montenegro, Olha Shkaravska, Marko Van Eekelen and Ricardo Peña. *Interpolation-based height analysis for improving a recurrence solver*. .......................................................... 95

Rody Kersten and Marko Van Eekelen. *Ranking Functions for Loops with Disjunctive Exit-Conditions*. .................................................. 111
Preface

The 2nd International Workshop on *Foundational and Practical Aspects of Resource Analysis* (FOPARA 2011) was held in Madrid, Spain, on May 19th, 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, FOPARA was co-located with the 12th International Symposium on *Trends in Functional Programming* (TFP) held in the same premises from May 16th to May 18th. This collocation was profitable for the two symposiums since a number of people registered to both events.

The workshop serves as a forum for presenting original research results that are relevant to the analysis of resource (time, space and others) consumption by computer programs. The workshop aims to bring together the researchers that work on foundational issues with the researchers that focus more on practical results. Therefore, both theoretical and practical contributions are encouraged. Also papers that combine theory and practice are welcome. The first workshop of this series FOPARA09 was organised by the Radboud University of Nijmegen at Eindhoven (The Netherlands) on November 2009.

There was a screening phase which selected works for the symposium. The screening acceptance is usually based on the requirement that the work should be in scope and contain relevant information for the FOPARA audience. This year, 9 works were submitted all of them accepted for presentation. After the workshop there will be a formal peer reviewing process which will select a subset of these works for publication in the Springer series *Lecture Notes in Computer Science*.

The scientific program also included an invited talk by Reinhard Wilhelm, from *Universitat des Saarlandes*, on *Timing Analysis and Timing Predictability*. 
Acknowledgements

We thank all the speakers, the authors, and the rest of participants for contributing to the success of FOPARA 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 FOPARA 2009 organizers Marko van Eekelen and Olha Shkaravska, 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
FOPARA 2011 Program Committee Chair
FOPARA 2011 Organizing Committee Chair

Madrid, May 2011
Program Committee

Purificaci´on Arenas  Complutense University of Madrid, ES
David Aspinall  University of Edinburgh, UK
David Cachera  IRISA/École normale supérieure de Cachan, FR
Marko van Eekelen  Radboud University Nijmegen and Open University, NL
Kevin Hammond  University of St. Andrews, UK
Martin Hofmann  LMU, Munich, DE
T´amas Kozsik  E¨otv¨os Lor´and University of Budapest, HU
Hans-Wolfgang Loidl  Heriot-Watt, Edinburgh, UK
Jean-Yves Marion  Loria, Nancy, FR
Simone Martini  University of Bologna, IT
Ricardo Pe˜na (PC Chair)  Complutense University of Madrid, ES
Simona Ronchi della Rocca  University of Turin, IT
Olha Shkaravska  Radboud University Nijmegen, NL

Local Organizing Committee

Ricardo Pe˜na (Chair)  Complutense University of Madrid, ES
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
A Higher Order Characterization of Probabilistic Polynomial Time

Ugo Dal Lago    Paolo Parisen Toldin

Abstract

We present RSLR, an implicit higher order characterization of the class \textbf{PP} of those problems which can be decided in probabilistic polynomial time with error probability smaller than \(1/2\). Analogously, we can get a characterization of the class \textbf{BPP}. RSLR is an extension of Hofmann’s SLR with a probabilistic primitive. We prove that this system enjoys subject reduction. Polytime soundness is obtained by syntactical means, as opposed to the standard literature on SLR-derived systems, which use semantics in an essential way.

1 Introduction

Implicit Computational Complexity (ICC) combines computational complexity, logic, and formal systems to give a machine independent account of complexity phenomena. It has been successfully applied to the characterization of a variety of complexity classes, especially in the sequential and parallel modes of computation (e.g., \textbf{FP}, \textbf{PSPACE}, \textbf{LOGSPACE}, \textbf{NC}). Its techniques, however, may be applied also to non-standard paradigms, like quantum computation [5] and concurrency [3].

Randomized computation is central to several areas of theoretical computer science, including cryptography, analysis of computation dealing with uncertainty, incomplete knowledge agent systems. In the context of computational complexity, probabilistic complexity classes like \textbf{BPP} are nowadays considered as very closely corresponding to the informal notion of feasibility, since a solution to a problem in \textbf{BPP} can be computed in polynomial time up to any given degree of precision.

Probabilistic polynomial computations, seen as oracle computations, were showed to be amenable to implicit techniques since the early days of ICC, by a relativization of Bellantoni and Cook’s safe technique [1]. They were then studied again in the context of formal systems for security, where probabilistic polynomial computations play a major role [6, 9]. These two systems build on Hofmann’s work on SLR, adding a random choice operator to the calculus. The system in [6], however, lacks higher-order recursion, and in both papers the characterization of the probabilistic classes is obtained by semantical means. While this is fine for completeness, we think it is not completely satisfactory for soundness—we know from the semantics that for any term of a suitable type its normal form \textit{may be} computed in the given bound, but no notion of evaluation is given for which computation time is guaranteed to be bounded.
In this paper we propose RSLR, another probabilistic variant of SLR, and we show that it characterizes the class PP of those problems which can be solved in polynomial time by a Turing Machine with error probability smaller than 1/2. This requires proving that any term in the language can be reduced in polynomial time, but also that any problems in PP can be represented in RSLR. A similar result will be proved for BPP. Unlike [6], RSLR has higher order recursion. Unlike [6] and [9] the bound on reduction time is obtained by syntactical means, giving an explicit notion of reduction which may realize that bound.

### 1.1 Related Works

We discuss here in more details the relations of our system to the previous work we already cited.

Mitchell, Mitchell, and Scedrov’s [6, 7] introduce OSLR, a type system that characterizes oracle polynomial time functionals. Even if inspired by SLR [4], OSLR does not admit primitive recursion on higher-order types, but only on ground types. The main theorem shows that terms of $\Box N^m \rightarrow N^n \rightarrow N$ define precisely the oracle polynomial time functionals, which is a class related but different from the ones we are interested in here. Finally, inclusion in the polynomial time class is proved without studying reduction from an operational view, but only via semantics: it is not clear for which notion of evaluation, computation time is guaranteed to be bounded.

Zhang’s [9] introduces a further system (CSLR) which builds on OSLR and allows higher-order recursion. The main interest of the paper are applications to the verification of security protocols. It is stated that CSLR defines exactly those functions that can be computed by probabilistic Turing machines in polynomial time, via a suitable variation of Hofmann’s techniques as modified by Mitchell et al. This is again a purely semantical proof.

Finally, both works are derived from Hofmann’s one, and as a consequence they both have potential problems with subject reduction. Indeed, as Hofmann showed in his work, subject reduction does not hold in SLR.

### 1.2 RSLR: An Informal Account

Our system is called RSLR, which stands for Random Safe Linear Recursion.

RSLR can be thought of as the system obtained by endowing SLR with a random primitive and restricting it, to be able to prove some operational form of polynomial time soundness. The final result is indeed quite similar to (a probabilistic variation of) Bellantoni, Niggl and Schwichtenberg calculus RA [2, 8].

Actually, the only needed restriction with respect to SLR deals with linearity: keeping the size of reducts under control during normalization is very difficult in presence of higher-order duplication. For this reason, the two function spaces $A \rightarrow B$ and $A \rightarrow B$ collapse to just one in RSLR. This allows us to get a reasonably simple system for which polytime soundness can be proved explicitly, by studying the combinatorics of a reduction strategy. More specifically, terms of RSLR are shown to be reducible in polynomial time using a mixed strategy, where arguments of ground types are handled as in call-by-value, while those of higher-order types are passed to functions without being evaluated.
2 The Syntax of RSLR

RSLR is a fairly standard Curry-style lambda calculus with constants for the natural numbers, branching and recursion. Its type system, on the other hand, is based on ideas coming from linear logic (some variables can appear at most once in terms) and by a distinction between modal and nonmodal variables.

Let us introduce the category of types first:

Definition 2.1 (Types). The types of RSLR are generated by the following grammar:

\[ A ::= N \mid □A \rightarrow A \mid ■A \rightarrow A. \]

Types different from N are denoted with metavariables like H or G.

There are two function spaces in our system. Programs which can be typed as □A → B are such that the result in B can be computed in time somehow independent on the size of the argument (of type A) On the other hand, computing the result of functions in □A → B requires polynomial time in the size of their argument.

A notion of subtyping is used in RSLR to capture the intuition above by stipulating that the type □A → B is a subtype of □A → B.

Definition 2.2 (Aspects). An aspect is either □ or ■. Aspects are partially ordered by the relation < such that □ < ■.

Subtyping rules are in Figure 1.

![Subtyping Rules](image)

Figure 1: Subtyping rules.

Definition 2.3 (Terms). Terms and constants are defined as follows:

\[ t ::= x \mid c \mid ts \mid λx : A.t \mid \text{case}_{A}t \text{ zero } s \text{ even } r \text{ odd } q \mid \text{recursion}_{A}t s r; \]
\[ c ::= n \mid S_{0} \mid S_{1} \mid P \mid \text{rand}. \]

where x ranges over a denumerable set of variables and n ranges over the natural numbers seen as constants of base type.

As already mentioned, our notion of reduction is mixed: arguments of base type N are evaluated before being passed to functions, while arguments of higher order are passed to functions without necessarily evaluate them, in a call-by-name fashion. Before doing that, let’s define the one-step reduction relation:

Definition 2.4 (Reduction). The one-step reduction relation → is a ternary relation between terms, values in R\([0,1]\) and terms. If \((t, p, s) \in →\), we write \(t \rightarrow_{p} s\). If \(t \rightarrow_{1} s\), we write \(t \rightarrow s\), simply. The relation → is defined by the axioms in Figure 2 and the inference rules in Figure 4.
\[
\text{case}_\tau 0 \text{ zero } t \text{ even } s \text{ odd } r \rightarrow t
\]
\[
\text{case}_\tau (S_0 n) \text{ zero } t \text{ even } s \text{ odd } r \rightarrow s(S_0 n)
\]
\[
\text{case}_\tau (S_1 n) \text{ zero } t \text{ even } s \text{ odd } r \rightarrow r(S_1 n)
\]
\[
\text{recursion}_\tau 0 g f \rightarrow g
\]
\[
\text{recursion}_\tau n g f \rightarrow fn(\text{recursion}_\tau \left\lfloor \frac{n}{2} \right\rfloor g f)
\]
\[
S_0 n \rightarrow 2 \cdot n
\]
\[
S_1 n \rightarrow 2 \cdot n + 1
\]
\[
P0 \rightarrow 0
\]
\[
Pn \rightarrow \left\lfloor \frac{n}{2} \right\rfloor
\]
\[
(\lambda x : N.t)n \rightarrow t[x/n]
\]
\[
(\lambda x : H.t)s \rightarrow t[x/s]
\]
\[
\text{rand} \rightarrow \frac{1}{2} 0
\]
\[
\text{rand} \rightarrow \frac{1}{2} 1
\]

Figure 2: Reduction Steps: Axioms

\[
\begin{array}{c}
t \rightarrow_p s \\
t \rightarrow_p s \\
tr \rightarrow_p sr \\
rt \rightarrow_p rs \\
\lambda x : A.t \rightarrow_p \lambda x : A.s
\end{array}
\]

Figure 3: Reduction Steps: Inference Rules
The reduction relation → only models one-step reduction sequences. To

**Definition 2.5 (Multistep Reduction).** The ternary relation \( \rightsquigarrow \) between terms, probabilities and terms is defined in Figure 4. As usual, \( \rightsquigarrow \) is syntactic sugar for \( \rightsquigarrow_1 \).

![](image)

Figure 4: Reduction Steps: Inference Rules

We are finally able to present the type system. Preliminary to that is the definition of a proper notion of a context.

**Definition 2.6 (Contexts).** A context \( \Gamma \) is a finite set of assignment of types and aspects to variables, in the form \( x : a \cdot A \). The union of two disjoint contexts \( \Gamma \) and \( \Delta \) is denoted as \( \Gamma \cup \Delta \). In doing so, we implicitly assume that the variables in \( \Gamma \) and \( \Delta \) are pairwise distinct. The union \( \Gamma \cup \Delta \) is sometimes denoted as \( \Gamma ; \Delta \): this way we want to stress that all types appearing in \( \Gamma \) are ground. With the expression \( \Gamma \vdash x : \Gamma(x) \) we mean that any aspect \( b \) appearing in \( \Gamma \) is such that \( b \vdash \Gamma(x) \).

![](image)

Figure 5: Type rules

### 3 The Main Results

A first-order term of arity \( k \) is a closed, well typed term of type \( a_1 \cdot N \rightarrow a_2 \cdot N \rightarrow \ldots \rightarrow a_k \cdot N \rightarrow N \) for some \( a_1, \ldots, a_k \).
The most difficult (and interesting!) result about RSLR is definitely polytime soundness: every (instance of) a first-order term can be reduced to its normal form in a polynomial number of steps using the previously introduced notion of reduction. Polytime soundness can be proved, following [2], by showing that:

- Any term of base type which does not contain any recursion can be reduced to its normal form with very low time complexity.
- Any term (non necessarily of base type!) can be reduced to another not containing any recursion in polynomial time.

By gluing these two results together, we obtain what we need, namely an effective and efficient procedure to compute the normal forms of terms. Formally:

**Theorem 3.1** (Soundness). Suppose $t$ is a first order term of arity $k$. Then there is a well-defined complete strategy of reduction and a polynomial $p_t$ such that if $t n_1 n_2 \ldots n_k \sim_p s$, then $k, |s| \leq p_t(\sum_{i=1}^k |n_i|)$.

What about the converse? Clearly, we can proceed as in [6] and prove that the class of probabilistic functionals which are representable in RSLR equals the class of those probabilistic functionals which can be computed in polynomial time. Here, however, we are more interested in the correspondence with complexity classes, which are classes of subsets of the natural numbers.

**Definition 3.1.** Let $t$ a first-order term of arity 1 and let $p \in \mathbb{R}_{[0,1]}$. Then $t$ is said to $p$-represents a language $L \subseteq \mathbb{N}$ iff:

1. If $n \in L$, then $t \sim_q 0$ where $q > p$.
2. If $n \notin L$, then $t \sim_q m$ where $m \neq 0$ and $q > p$.

Theorem 3.1, together with an encoding of probabilistic Turing Machines into RSLR allows us to conclude that:

**Theorem 3.2** (Completeness for PP). The set of languages which can be $p$-represented in RSLR for some $1/2 \leq p \leq 1$ equals PP.

So, in a sense, RSLR is strongly linked with the complexity class PP.

But, interestingly, we can go beyond and capture a more interesting complexity class:

**Theorem 3.3** (Completeness for BPP). The set of languages which can be $p$-represented in RSLR for some $2/3 \leq p < 1$ equals BPP.

**References**


Typing a Core Binary Field Arithmetic in a Light Logic

Emanuele Cesena\textsuperscript{1}\textsuperscript{*}, Marco Pedicini\textsuperscript{2}, and Luca Roversi\textsuperscript{3}

\textsuperscript{1} Politecnico di Torino, Dip. di Automatica e Informatica, Torino, Italy  
emanuele.cesena@polito.it
\textsuperscript{2} Istituto per le Applicazioni del Calcolo “Mauro Picone”, CNR, Roma, Italy  
marco@iac.cnr.it
\textsuperscript{3} Università degli Studi di Torino, Dip. di Informatica, Torino, Italy  
roversi@di.unito.it

Abstract. We design a library for binary field arithmetic and we supply a core API which is completely developed in DLAL, extended with a fix point formula. Since DLAL is a restriction of linear logic where only functional programs with polynomial evaluation cost can be typed, we obtain the core of a functional programming setting for binary field arithmetic with \textit{built-in} polynomial complexity.

1 Introduction

Embedded systems (smart cards, mobile phones, sensors) are very heavy on resources. Low memory and computational power force programmers to choose specific algorithms and fine tune them in order to carefully manage the space and time complexity. There is an applicative domain where these constraints on resources cause serious difficulties: the implementation of cryptographic primitives, that is the foundation for strong security mechanisms and protocols.

We have started reasoning about a controlled programming setting, that should enable the certification of resource usage (memory and computation time), in a functional programming language. We are aware of different approaches to solve analogous problems, for instance the Computer Aided Cryptography Engineering (CACE) European project\textsuperscript{4} whose mission is “to enable verifiable secure cryptographic software engineering to non-experts by developing a toolbox which automatically produces high-performance solutions from natural specifications”.

What if these difficulties were \textit{automagically} overcome by imposing an appropriate type discipline to the programming language?

This paper describes preliminary results on how to devise a programming language that grants a natural programming style in the implementation of specific number theoretic algorithms, in combination with a type discipline which ensures time/space computational bounds. More precisely, we investigate how to achieve certain outcomes developed in the area of Implicit Computational Complexity (ICC). ICC mainly aims to research strong mathematical roots for computational complexity theory. One approach to ICC, whose roots are found in proof theory, focuses on extracting functional language

\textsuperscript{*} Partially supported by the European project TClouds (http://www.tclouds-project.eu).
\textsuperscript{4} http://www.cace-project.eu
primitives by studying the meaning of “developing a computation” inside mathematical structures which, in turn, can be inductively described by deductive systems. We focus on DLAL [1], a deductive system derived from linear logic, whose formulas can be types of \( \lambda \)-terms, such that if a \( \lambda \)-term \( M \) has a formula of DLAL as type, then \( M \) reduces to its normal form in a time which is a polynomial in the dimension of \( M \).

We propose to put his theory into practice by developing and implementing a core library of combinators, namely \( \lambda \)-terms, typeable in DLAL. The library currently implements a subset of functionalities which are needed for binary field arithmetic (cf., e.g., [2, Section 11.2]). The practical relevance of completing such a library is to import functional programming technology with a known predetermined complexity into the area of applied cryptography.

**Contributions.** Defining a core library that correctly implements finite field arithmetic is a result in itself. The reason is that when programming non obvious combinators typeable in DLAL, the main obstacle lies in the application of the standard divide-et-impera paradigm: first split the problem into successively simpler ones until the solution becomes trivial, then compose the results. Composition is the harmful activity as soon as we face complexity issues. For example, using the output of a sub-problem, which results from an iteration, as the input of another iteration may yield a computational complexity blowup. This is why, in DLAL, naively manipulating lists by means of iterations, can rapidly “degrade” to situations where compositions which would be natural in standard \( \lambda \)-calculus simply get forbidden. It is for this reason that \( \lambda \)-terms in DLAL which implement the low level library with finite field operations are not the natural ones that we could write using \( \lambda \)-calculus typeable in System F.

To overcome the need of programming with non natural \( \lambda \)-terms, we follow [3], which promotes standard programming patterns to assure readability and soundness of functional programs. We build an experimental API on top of our library, which exports standard programming patterns. The goal of supplying an API is to help non experts writing \( \lambda \)-terms which are not directly typeable in DLAL, but which, roughly speaking, can be checked to compile into \( \lambda \)-terms with a type in DLAL.

**Related Works on Polynomial Time Languages.** A programming language inspired by Haskell is described in [4]. The programs that can be developed in it belong to the class of polynomial time functions because the language inherits the principles of the \( \lambda \)-terms, or, equivalently, of the proof-nets, of LAL [5]. However, we are not aware of any attempt to exploit it to program libraries with a real potential impact. The approach of [4] to the development of a real programming language for polynomial time computations is quite orthogonal to ours. We started bottom-up, showing that a reasonably interesting library can be developed inside DLAL. Then, we attempted to import standard programming patterns which were compatible with the typing discipline of DLAL. In [4], the language is given with the assumption that its primitives will really be used.

The same occurs in [6] and [7]. The former extends \( \lambda \)-calculus to give formulas of SLL [8]. The latter introduces POLA, a programming language which mixes object oriented and recursion schemes for which an interpreter is also available\(^5\). The best

\(^5\) [http://projects.wizardlike.ca/projects/pola](http://projects.wizardlike.ca/projects/pola)
developed project we are aware of, and which brings theoretical results related to the world of polynomial time bounded functions “down to” the practical level, is based on [9, 10]. The language exploits formulas of a smartly crafted version of multiplicative linear logic as types and is based on recursion schemes à la System T. We are still far from those levels of migration of theory to practice.

Our main distinguishing feature is that we remain loyal to the theoretical properties of DLAL, while allowing programming with standard patterns of functional programming.

2 Typed Functional Assembly

\(\lambda\)-calculus. Given a set \(\mathcal{V}\), which we range over by any lowercase Latin letters, the set \(\Lambda\) of \(\lambda\)-terms, which we range over by the uppercase Latin letter \(M, N, P, Q, R\), contains the terms generated as follows:

\[
M ::= \mathcal{V} | \lambda x.M | (M) M
\]

The set of free variables in \(M\) is \(fv(M)\). The number of occurrences of a free variable \(x\) in a term \(M\) is \(#(x, M)\). The set \(\Lambda^v\) of values of our computations, which we range over by the uppercase Latin letters \(V, W, X\), is defined as follows:

\[
V ::= \mathcal{V} | \lambda x.V | (x) V
\]

We remark that \(\Lambda^v\) coincides the standard \(\beta\) normal forms.

![Big steps rewriting relation \(\Downarrow\) on \(\Lambda\) with results in \(\Lambda^v\)](image)

**Big Steps Rewriting Relation on \(\lambda\)-calculus.** The relation \(\Downarrow \subset \Lambda \times \Lambda^v\) is inductively defined in Figure 1. We denote the size of a derivation whose conclusion is \(M \Downarrow V\) by \(|M \Downarrow V|\); that is the number of rule instances that occur in the derivation.

2.1 Type assignment

We introduce a type assignment \(TFA\) which gives formulas of Linear Logic as types to \(\lambda\)-terms. \(TFA\) is DLAL [1] whose set of formulas is quotiented by a specific recursive equation. We recall that adding a recursive equation among the formulas does not negatively affect polynomial time soundness of DLAL normalization which only depends on the structural constraints that the process of building the formulas puts on the form of the derivations [11].
Types for TFA. Given a set $G$ of formula variables, which we range over by lowercase Greek letters, the set $F$ of formulas, that we range over by the uppercase Latin letters $A, B, C, D$, is defined as follows:

$$A ::= \alpha \mid A \rightarrow A \mid !A \rightarrow A \mid \forall \alpha. A \mid \$A$$

Note that modal formulas $!A$ can occur in negative positions only.

We obtain the set of types $T$ when we consider the quotient of $F$ by the following fix-point equation:

$$S \equiv \forall \alpha. S[\alpha] \quad (3)$$

where $S[\alpha] \equiv (B_2 \rightarrow \alpha) \rightarrow ((B_2 \otimes S) \rightarrow \alpha) \rightarrow \alpha$ and $B_2$ is defined in Figure 3. We say $S$ is the type of Sequences. Thus, we actually use formulas which are equivalence classes of types in $T$.

Every time we use $S$ as type of a $\lambda$-term $M$, we can equivalently use any of its “unfolded forms” as type of $M$ as well. Figure 3 also introduces relevant types we shall use to develop our first level library. As a notation, $A[\beta/\alpha]$ is the clash free substitution of $B$ for every free occurrence of $\alpha$ in $A$.

Type assignment TFA. The type assignment system TFA is in Figure 2. It proves judgments $\Delta \vdash \Gamma \vdash M : A$. Every context $\Delta$ is exponential, while $\Gamma$ linear. Every context is a finite domain function $x_1 : A_1 \ldots x_n : A_n$ with domain $\{x_1 \ldots x_n\}$, and co-domain $\{A_1 \ldots A_n\}$. Every pair $x : A$ of any kind of context is a type assignment for a variable. The size $|\cdot|$ of a context is the number of elements in its domain.

Tuples as primitives. The definition of tuples in Figure 3 supports the introduction of the tuples as primitives, as follows. Extending $\lambda$-calculus with tuples means adding the following clauses to (1):

$$M ::= \ldots | \langle M \ldots M \rangle | \lambda(x \ldots x).M \quad (4)$$
<table>
<thead>
<tr>
<th>Type</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite types</td>
<td>$B^\alpha_n \equiv \alpha \rightarrow \cdots \alpha \rightarrow \alpha$</td>
</tr>
<tr>
<td></td>
<td>$B_n \equiv \forall \alpha. B^\alpha_n$</td>
</tr>
<tr>
<td>Tuples</td>
<td>$(A_1 \otimes \cdots \otimes A_n)[\alpha] \equiv A_1 \rightarrow \cdots \rightarrow A_n \rightarrow \alpha$</td>
</tr>
<tr>
<td></td>
<td>$(A_1 \otimes \cdots \otimes A_n) \equiv \forall \alpha. (A_1 \otimes \cdots \otimes A_n)[\alpha] \rightarrow \alpha$</td>
</tr>
<tr>
<td>Church numerals</td>
<td>$U[\alpha] \equiv !(\alpha \rightarrow \alpha) \rightarrow $$(\alpha \rightarrow \alpha)$</td>
</tr>
<tr>
<td></td>
<td>$U \equiv \forall \alpha. U[\alpha]$</td>
</tr>
<tr>
<td>Lists</td>
<td>$L(A)[\alpha] \equiv !(A \rightarrow \alpha \rightarrow \alpha) \rightarrow $$(\alpha \rightarrow \alpha)$</td>
</tr>
<tr>
<td></td>
<td>$L(A) \equiv \forall \alpha. L(A)[\alpha]$</td>
</tr>
<tr>
<td>Church words</td>
<td>$L_2 \equiv L(B_2)$</td>
</tr>
</tbody>
</table>

**Fig. 3.** Relevant (defined) types

So, values in (2) also include:

$$V ::= \ldots \mid \langle V \ldots V \rangle$$  \hspace{1cm} (5)

and the set of rules in Figure 1 must contain:

$$\frac{M_1 \Downarrow V_1 \ldots M_n \Downarrow V_n}{\langle M_1 \ldots M_n \rangle \Downarrow \langle V_1 \ldots V_n \rangle}$$ \hspace{1cm} (p)

$$\frac{M \Downarrow \lambda(x_1 \ldots x_n).V \quad N \Downarrow \langle V_1 \ldots V_n \rangle \quad V[V_1/x_1 \ldots V_n/x_n] \Downarrow W}{(M)N \Downarrow W}$$ \hspace{1cm} (@p)

Finally, we add the following derivable rules to those ones in Figure 2:

$$\frac{\Delta_1 \mid \Gamma_1 \vdash M_1 : A_1 \quad \ldots \quad \Delta_n \mid \Gamma_n \vdash M_n : A_n}{\langle \Delta_1 \ldots \Delta_n \mid \Gamma_1 \ldots \Gamma_n \vdash \langle M_1 \ldots M_n \rangle : (A_1 \otimes \cdots \otimes A_n) \rangle \Downarrow 1 \otimes I}$$

$$\frac{\Delta \mid \Gamma, x_1 : A_1 \ldots x_n : A_n \vdash M : B}{\Delta \mid \Gamma \vdash \lambda(x_1 \ldots x_n).M : (A_1 \otimes \cdots \otimes A_n) \rightarrow B}$$ \hspace{1cm} ($\rightarrow 1 \otimes$)

Saying that the here above rules are derivable means that we use tuple as abbreviations, as follows:

$$\langle M_1 \ldots M_n \rangle \equiv \lambda x.((x) M_1) \ldots) M_n$$  \hspace{1cm} (6)

$$\lambda(x_1 \ldots x_n).M \equiv \lambda p.(p) \lambda x_1 \ldots \lambda x_n .M$$  \hspace{1cm} (7)
3 A Library for Binary Field Arithmetic

In this section, we present our library for the arithmetic in binary fields written in DLAL. The library is organized in layers, as shown in Figure 4.

The lowest layer contains basic definitions and it is interpreter-specific. We have currently implemented the library with LCI⁶, which is an interpreter for pure λ-calculus. We thus needed to define basic types, such as Church words, or DLAL-specific combinators.

The core library layer contains all the combinators to work on basic types. We put particular care in the definition of common functional-programming patterns in DLAL, and to reuse them, whenever possible, while defining other combinators.

Finally, in the binary fields arithmetic layer we group all the combinators related to operations over binary polynomials, like sum, multiplication and modular reduction.

In future work, we plan to extend the library by implementing other layers, such as arithmetic of elliptic curves or other cryptographic primitives, on top of the binary field arithmetic layer.

In the following subsections we present type and behaviour of the relevant combinators, while the full definition as λ-term is in Appendix A.

3.1 Basic Definitions and Types

Figure 5 gives the names to relevant formulas which become our relevant types and identifies the λ-terms that we use as canonical values of the corresponding type. In every Sequence \([b_{n-1} \ldots b_0]\) and Church word \(\{b_{n-1} \ldots b_0\}\) the least significant bit (l.s.b.) is \(b_0\), while the most significant bit (m.s.b.) is \(b_{n-1}\).

In DLAL we can derive the rule paragraph lift:

\[
\emptyset \mid \emptyset \vdash M : A \rightsquigarrow B \quad \emptyset \mid \emptyset \vdash \§[M] : §A \rightsquigarrow §B \quad \§L
\]

where \(\§[M] \equiv \lambda x. (M)\ x\) is the paragraph lift of \(M\). As obvious generalization, \(n\) consecutive applications of \(\§L\) define \(\§^n[M] \equiv \lambda x. (\ldots \lambda x. (M)\ x\ldots)\ x\), that contains \(n\) nested \(\§[\cdot]\). Its type is \(\§^n A \rightsquigarrow \§^n B\). Borrowing terminology from proof nets, the application of \(n\) paragraph lift of \(M\) embeds it in \(n\) paragraph boxes, leaving the behaviour of \(M\) unchanged:

\[
(\§^n[M])\ N \Downarrow (M)\ N
\]

⁶ http://lci.sourceforge.net
<table>
<thead>
<tr>
<th></th>
<th>(typed) Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booleans</td>
<td>1 ≡ λxyz.x : B₂</td>
</tr>
<tr>
<td></td>
<td>θ ≡ λxyz.y : B₂</td>
</tr>
<tr>
<td></td>
<td>⊥ ≡ λxyz.z : B₂</td>
</tr>
<tr>
<td>Church numerals</td>
<td>uε ≡ λfx.x : U</td>
</tr>
<tr>
<td></td>
<td>(\vec{n} ≡ \lambda f x. (f) . . . (f) \ x : U)</td>
</tr>
<tr>
<td>Church words</td>
<td>{ε} ≡ λfx.x : L₂</td>
</tr>
<tr>
<td></td>
<td>[bₙ₋₁ . . . b₀] ≡ (\lambda f x. ((f) bₙ₋₁) . . . ((f) b₀) \ x : L₂)</td>
</tr>
<tr>
<td>Sequences</td>
<td>[ε] ≡ (\lambda tc.(t) \downarrow : S)</td>
</tr>
<tr>
<td></td>
<td>[bₙ₋₁ . . . b₀] ≡ (\lambda tc.(c) \ (bₙ₋₁, [bₙ₋₂ . . . b₀]) : S)</td>
</tr>
</tbody>
</table>

**Fig. 5.** Canonical values of data-types

The combinator bCast\(^m\) : B₂ \(\rightarrow\) \(\underbrace{\otimes \cdots \otimes}_{m+1} B₂\) embeds a boolean into \(m + 1\) paragraph boxes, without altering the boolean:

\[
\text{(bCast}^m) \ b \downarrow b
\]

The combinator bV\(_t\) : B₂ \(\rightarrow\) \(\underbrace{B₂ \otimes \cdots \otimes B₂}_{t}\), for every \(t \geq 2\), produces \(t\) copies of a boolean:

\[
\text{(bV}^t) \ b \downarrow \underbrace{\langle b . . . b \rangle}_{t}
\]

The combinator wSuc : B₂ \(\rightarrow\) L₂ \(\rightarrow\) L₂ implements the successor on words by adding a m.s.b. to a word:

\[
\text{(wSuc) b} \ [bₙ₋₁ . . . b₀] \downarrow \langle b \ bₙ₋₁ . . . b₀ \rangle
\]

The combinator wCast\(^m\) : L₂ \(\rightarrow\) \(\underbrace{\otimes \cdots \otimes}_{m+1} L₂\), for every \(m \geq 0\), embeds a word into \(m + 1\) paragraph boxes, without altering the structure of the word:

\[
\text{(wCast}^m) \ [bₙ₋₁ . . . b₀] \downarrow \langle bₙ₋₁ . . . b₀ \rangle
\]

The combinator wV\(_t\) \(\rightarrow\) \(\underbrace{L₂ \otimes \cdots \otimes L₂}_{t}\), for every \(t \geq 2, m \geq 0\), produces \(t\) copies of a word deepening the result into \(m + 1\) paragraph boxes:

\[
\text{(wV}^t) \ [bₙ₋₁ . . . b₀] \downarrow \langle [bₙ₋₁ . . . b₀] . . . [bₙ₋₁ . . . b₀] \rangle
\]
3.2 Core Library

Operations on Bits.
The combinator \( \text{Xor} : \mathbb{B}_2 \rightarrow \mathbb{B}_2 \rightarrow \mathbb{B}_2 \) extends the exclusive or as follows:

\[
\begin{align*}
((\text{Xor})0)0 & \downarrow 0 & ((\text{Xor})1)1 & \downarrow 0 \\
((\text{Xor})0)1 & \downarrow 1 & ((\text{Xor})1)0 & \downarrow 1 \\
((\text{Xor})\bot)b & \downarrow b & ((\text{Xor})b)\bot & \downarrow b \\
\end{align*}
\]

Whenever one argument is \( \bot \) then it gives back the other argument. This is an application oriented choice. Later we shall see why.

The combinator \( \text{And} : \mathbb{B}_2 \rightarrow \mathbb{B}_2 \rightarrow \mathbb{B}_2 \) extends the and as follows:

\[
\begin{align*}
((\text{And})0)0 & \downarrow 0 & ((\text{And})1)1 & \downarrow 1 \\
((\text{And})0)1 & \downarrow 0 & ((\text{And})1)0 & \downarrow 0 \\
((\text{And})\bot)b & \downarrow \bot & ((\text{And})b)\bot & \downarrow \bot \\
\end{align*}
\]

Whenever one argument is \( \bot \) then the result is \( \bot \). Again, this is an application oriented choice.

Operations on Sequences.
The combinator \( \text{sSpl} : \mathbb{S} \rightarrow (\mathbb{B}_2 \otimes \mathbb{S}) \) splits the sequence it takes as input in a pair with the m.s.b. and the corresponding tail:

\[
(\text{sSpl})[b_{n-1} \ldots b_0] \downarrow (b_{n-1}, [b_{n-2} \ldots b_0])
\]

Operations on Church Words.
The combinator \( \text{wRev} : \mathbb{L}_2 \rightarrow \mathbb{L}_2 \) reverses the bits of a word:

\[
(\text{wRev})\{b_{n-1} \ldots b_0\} \downarrow \{b_0 \ldots b_{n-1}\}
\]

The combinator \( \text{wDrop} \bot : \mathbb{L}_2 \rightarrow \mathbb{L}_2 \) drops all the (initial) occurrences\(^7\) of \( \bot \) in a word:

\[
(\text{wDrop} \bot)\{\bot \ldots \bot b_{n-1} \ldots b_0\} \downarrow \{b_{n-1} \ldots b_0\}
\]

The combinator \( \text{w2s} : \mathbb{L}_2 \rightarrow \mathbb{S} \) casts a word into a sequence:

\[
(\text{w2s})[b_{n-1} \ldots b_0] \downarrow [b_{n-1} \ldots b_0]
\]

The combinator \( \text{wProj} : \mathbb{L}(A \otimes B) \rightarrow \mathbb{L}(A) \) projects the first component of a word of couples:

\[
(\text{wProj})\{\langle a_{n-1}, b_{n-1} \rangle \ldots \langle a_0, b_0 \rangle\} \downarrow \{a_{n-1} \ldots a_0\}
\]

\(^7\) The current definition actually drops all the occurrences of \( \bot \) in a Church word, however we shall only apply \( \text{wDrop} \bot \) to words that contain \( \bot \) in the most significant bits.
Meta-combinators on Church Words. Meta-combinators are $\lambda$-terms with one or
two “holes” that allow to use standard higher-order programming patterns to extend
API. Holes must be filled with $\lambda$-terms having the right type. We shall discuss how
using meta-combinators effectively, after their introduction here below.

The first meta-combinator we deal with is $\text{Map}[:,]$. Let $F : A \rightarrow B$ be a closed term.
Then, $\text{Map}[F] : \mathbb{L}(A) \rightarrow \mathbb{L}(B)$ applies $F$ to every element of the word that $\text{Map}[F]$ takes as
argument, and yields the final word:

$$(\text{Map}[F]) \{b_{n-1} \ldots b_0\} \Downarrow \{b'_{n-1} \ldots b'_0\}$$

if $(F) b_i \Downarrow b'_i$, for every $0 \leq i \leq n - 1$.

The second meta-combinator is $\text{Fold}[\cdot, \cdot]$. Let $F : A \rightarrow B \rightarrow B$ and $S : B$ be closed terms.
Let also assume there is $\text{Cast}^0 : B \rightarrow \$B$. Then, $\text{Fold}[F, S] : \mathbb{L}(A) \rightarrow \$B$, starting from
the initial value $S$, iterates $F$ over the word taken as parameter and build up a return
value.

$$(\text{Fold}[F, S]) \{b_{n-1} \ldots b_0\} \Downarrow b'$$

if $((F) b_i) b'_i \Downarrow b'_{i+1}$, for every $0 \leq i \leq n - 1$, where we set $b'_0 = S$ and $b'_n = b'$.

The third meta-combinator is $\text{MapThread}[:,]$. Let $F : A \rightarrow B \rightarrow C$ be a closed term. Then,
$\text{MapThread}[F] : \mathbb{L}(A) \rightarrow \mathbb{L}(B) \rightarrow \mathbb{L}(C)$ applies $F$ to the elements of the words taken as
arguments, and yields the final word:

$$((\text{MapThread}[F]) \{a_{n-1} \ldots a_0\}) \{b_{n-1} \ldots b_0\} \Downarrow \{c_{n-1} \ldots c_0\}$$

if $((F) a_i) b_i \Downarrow c_i$, for every $0 \leq i \leq n - 1$.

In particular, $\text{MapThread}[:, \cdot] : \mathbb{L}(A) \rightarrow \mathbb{L}(B) \rightarrow \mathbb{L}(A \otimes B)$, given two words, creates a
word of couples:

$$((\text{MapThread}[:, \cdot]) \{a_{n-1} \ldots a_0\}) \{b_{n-1} \ldots b_0\} \Downarrow \{(a_{n-1}, b_{n-1}) \ldots (a_0, b_0)\}$$

The fourth meta-combinator is $\text{MapState}[:,]$. Let $F : (A \otimes S) \rightarrow (B \otimes S)$ be a closed term.
Then, $\text{MapState}[F] : \mathbb{L}(A) \rightarrow S \rightarrow \mathbb{L}(B)$ applies $F$ to the elements of the word taken as
argument, keeping track of a state (of type $S$) during the iteration.

$$((\text{MapState}[F]) \{b_{n-1} \ldots b_0\}) s_0 \Downarrow \{b'_{n-1} \ldots b'_0\}$$

if $(F) (b_i, s_i) \Downarrow (b'_i, s_{i+1})$, for every $0 \leq i \leq n - 1$.

3.3 Binary Field Arithmetic

Here we set how the standard mathematical notation that defines binary field arithmetic
gets mapped into $\lambda$-terms that exploit the programming patterns available from the core
library level. We start by recalling the essentials on binary field arithmetic. For wider
details we address the reader to [2, Section 11.2].

Let $f(X) \in \mathbb{F}_2[X]$ be an irreducible polynomial of degree $n$ over $\mathbb{F}_2$, and let $\beta \in \overline{\mathbb{F}}_2$ be
a root of $f(X)$ in the algebraic closure of $\mathbb{F}_2$. Then, the finite field $\mathbb{F}_{2^n} \cong \mathbb{F}_2[X]/(f(X)) \cong
\( \mathbb{F}_2(\beta) \). The set of elements \( \{1, \beta, \ldots, \beta^{n-1}\} \) is a basis of \( \mathbb{F}_{2^n} \) as a vector space over \( \mathbb{F}_2 \) and we can represent a generic element of \( \mathbb{F}_{2^n} \) as a polynomial in \( \beta \) of degree lower than \( n \):

\[
\mathbb{F}_{2^n} \ni a = \sum_{i=0}^{n-1} a_i \beta^i = a_{n-1} \beta^{n-1} + \cdots + a_1 \beta + a_0 , \quad a_i \in \mathbb{F}_2 .
\]

Moreover, the isomorphism \( \mathbb{F}_{2^n} \cong \mathbb{F}_2[X]/(f(X)) \) allows us to implement the arithmetic of \( \mathbb{F}_{2^n} \) relying on the arithmetic of \( \mathbb{F}_2[X] \) and reduction modulo \( f(X) \).

Since each element \( a_i \in \mathbb{F}_2 \) can be encoded as a bit, we can represent each element of \( \mathbb{F}_{2^n} \) as a Church word of bits of type \( \mathbb{L}_2 \).

In what follows, we shall denote by \( n \) the Church numeral representing \( n = \deg f(X) \), and by \( f \) the Church word:

\[
f \equiv \{ f_n \cdots f_0 \ \bot \cdots \bot \} ,
\]

where \( f_i \) are such that \( f(X) = \sum f_i X^i \). Note that \( f \) has length \( 2n - 1 \). The \( \bot \) in the least significative part are included for technical reasons, to simplify the discussion later.

**Addition.** Let \( a, b \in \mathbb{F}_{2^n} \). The addition \( a + b \) is computed component-wise, i.e., setting \( a = \sum a_i \beta^i \) and \( b = \sum b_i \beta^i \), then \( a + b = \sum (a_i + b_i) \beta^i \). The sum \( (a_i + b_i) \) is done in \( \mathbb{F}_2 \) and corresponds to the bitwise exclusive or. This led us to the following definition:

The combinator \( \text{Add} : \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n} \rightarrow \mathbb{F}_{2^n} \) is:

\[
\text{Add} \equiv \text{MapThread}[\text{Xor}]
\]

**Modular Reduction.**

The combinator \( \text{wMod}[n, f] : \mathbb{L}_2 \rightarrow \mathbb{F}_{2^n} \) is:

\[
\text{wMod}[n, f] \equiv \lambda d. (\text{wModEnd}) (n) (\lambda l. (\text{MapState}[\text{wModFun}]) l) (\bot, \emptyset) \text{wModBase}[d, f]
\]

where:

\[
\text{wModEnd} \equiv (\text{wDrop}\ \bot) (\text{wRev}) \text{wProj}
\]

\[
\text{wModFun} \equiv \lambda (e, s). (e) \lambda (d, f). (s) \lambda (s_0, s_1).
\]

\[
\left( \left( \left( \left( \left( s_0 \right) \right) \lambda f s_1. \lambda (f', f''). ((\left( (\text{Xor}) d \right) f', s_1), (1, f'')) (\text{wRev}_2) f) \right) \lambda f s_1. \left( \left( d, s_1 \right), (\emptyset, f) \right) \right) \right)
\]

\[
\lambda f s_1. \left( \left( d, (\bot, s_1) \right), (d, f) \right)
\]

\[
\lambda d f s_1. \left( \left( d, f \right), s_1 \right)
\]

\[
\text{wModBase}[d, f] \equiv ((\text{MapThread}[(, ,)]) (\text{wRev}) d) (\text{wRev}) f
\]
Square. Square in binary fields is a linear map (it is the absolute Frobenius automorphism). If $\mathbb{F}_{2^n} \ni a = \sum a_i \beta^i$, then $a^2 = \sum a_i \beta^{2i}$. This operation is obtained by inserting zeros between the bits that represent $a$ and leads to a polynomial of degree $2n - 2$, that needs to be reduced modulo $f(X)$.

Therefore, we introduce two combinators: $wSqr : \mathbb{L}_2 \to \mathbb{L}_2$ that performs the bit expansion, and $Sqr : \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$ that is the actual square in $\mathbb{F}_{2^n}$. We have:

$$Sqr \equiv (wMod(n,f))wSqr \quad (9)$$

and $wSqr \equiv \lambda l f x.((l) wSqrStep[f]) x$, where $wSqrStep[f] \equiv \lambda et.((f) \theta)((f) e) t$ has type $\mathbb{B}_2 \to \alpha \to \alpha$ if $f$ is a non linear variable with type $\mathbb{B}_2 \to \alpha \to \alpha$.

**Multiplication and Inversion.** Multiplication and inversion are under development.

For multiplication, we are considering the na"ive schoolbook method and possibly its extension to the comb method. The Karatsuba multiplication algorithm, that reduces the multiplication of $n$-bit words to multiplications of $n/2$-bit words, appears difficult to be translated in DLAL since it requires the concept of splitting a word in its half upper and lower parts.

For inversion, we are concentrating on the binary Euclidean algorithm, which is the "left-to-right" counterpart of the extended Euclidean algorithm (for a detailed analysis, we refer to Fong et al. [12]).

### 4 Developing (with) the Library

Beside the implementation of the library, we experimented the use of higher-order combinators to improve the readability of the code, as well as the programming experience. Inspired by [3], we have rewritten some combinators relying on standard programming pattern such as $\text{Map}[-]$ and $\text{Fold}[-,-]$, “simulating” the behavior of a programmer that wants to add new functionality to the library. The idea is to let the programmer write a combinator in a more comfortable style, and then to compile the combinator to a value that admits a type in DLAL. In the following, we give some relevant examples of increasing difficulty.

We know that $w2s$ is defined as $w2s \equiv \lambda l.((l) \lambda estc.(c) \langle e, s \rangle) [e]$. A programmer could anyway define it by using the standard programming pattern $\text{Fold}[-,-]$ as follows:

$$w2sFromFold \equiv \text{Fold}[\lambda estc.(c) \langle e, s \rangle, [e]]$$

The combinator $w2sFromFold$ is a legal one because $w2sFromFold$ compiles exactly to $w2s$. The compilation consists of in-line substituting the parameters of $\text{Fold}[-,-]$ and of applying the rewriting steps in Figure 1, whose key intermediate $\lambda$-terms are $\lambda l.((l) \lambda ez.\lambda tc.(c) \langle e, z \rangle) e [e]$ and $\lambda l.((l) \lambda estc.(c) \langle e, z \rangle) [e]$.

As a second example, we consider the combinator $wProj \equiv \lambda l f x.((l) \lambda (a, b).(f) a) x$, we define the following combinator and we show that it is equivalent to the above one:

$$wProjFromMap \equiv \text{Map}[\lambda (a, b).a]$$
We recall that \( \text{wProj} : \mathbb{L}(A \otimes B) \rightarrow \mathbb{L}(A) \). This assumption, i.e., each element \( e \) of the input word is \( \langle a', b' \rangle \) for some \( a' : A, b' : B \), is needed while compiling the above expression. The key step is the reduction from \( \lambda f. x.((l) \lambda e. (f) (\lambda (a,b).a) e) x \) to \( \lambda f. x.((l) \lambda (a', b'), (f) (\lambda (a,b).a) (a', b')) x \), by replacing \( \langle a', b' \rangle \) for \( e \) in accordance with the assumption.

Finally, we show that the combinator \( \text{Map}[F] \equiv \lambda f. x.((l) \lambda e. (f) (F) e) x \) can be written using \( \text{Fold}[-,-] \) (see also [3, Section 2]) as:

\[
\text{MapFromFold}[F] \equiv \text{Fold}[\lambda e p f. x.((f) (F) e)((p) f) x, [\varepsilon]]
\]

Here, the compilation process shows that \( \text{Map}[F] l' \) and \( \text{MapFromFold}[F] l' \) are equivalent to the same value. We proceed by induction on the length of the Church word \( l' \). First, we note that:

\[
\text{MapFromFold}[F] \Downarrow \lambda l.((l) \lambda e p f. x.((f) (F) e)((p) f) x) [\varepsilon]
\]

The base case is easy to check: \( (\text{Map}[F]) [\varepsilon] \Downarrow [\varepsilon] \) and \( (\text{MapFromFold}[F]) [\varepsilon] \Downarrow [\varepsilon] \).

We now prove the inductive case. Let \( l \equiv \lambda f. x.((f) b_{n-1}) \ldots ((f) b_0) x \) be a Church word of length \( n \). Assume that \( (\text{Map}[F]) l \Downarrow V \) and \( (\text{MapFromFold}[F]) l \Downarrow V \). We want to show that \( \text{Map}[F] \) and \( \text{MapFromFold}[F] \) reduce to the same term for a Church word \( l' \equiv \lambda f. x.((f) b) ((l) f) x \) of length \( n + 1 \). We report the key intermediate \( \lambda \)-terms:

\[
(\text{Map}[F]) l' \Downarrow \lambda f. x.((l') \lambda e. (f) (F) e) x
\]

ind. hyp. \( \Downarrow \lambda f. x.((f) (F) b) ((l) \lambda e. (f) (F) e) x \quad (10) \)

\[
(\text{MapFromFold}[F]) l' \Downarrow ((l') \lambda e p f. x.((f) (F) e)((p) f) x) [\varepsilon]
\]

ind. hyp. \( \Downarrow ((\lambda e p f. x.((f) (F) e)((p) f) x) b) ((l) \lambda e p f. x.((f) (F) e)((p) f) x) [\varepsilon] \)

\[
\Downarrow (\lambda e p f. x.((f) (F) e)((p) f) x) b) V
\]

ind. hyp. \( \Downarrow \lambda f. x.((f) (F) b) ((V) f) x \)

This example is particularly relevant because \( \text{MapFromFold}[F] : \mathbb{L}(A) \rightarrow \mathbb{L}(B) \), and \( \text{Map}[F] : \mathbb{L}(A) \rightarrow \mathbb{L}(B) \) compile to a common term despite their types differ. This is possible by applying two \( \beta \)-expansions from (10) to (11) which do not duplicate any structure (see Appendix A.1.)

5 Conclusion and Future Work

We have presented a core library for binary field arithmetic developed DLAL. The main motivation behind this work is to achieve a programming framework with \textit{built-in} polynomial complexity and, from this perspective, this library is just a starting point. First, because it lacks multiplication and inversion. Then, because it lacks a complete realistic applicative example, as it could be elliptic curves cryptography. In the same line, the implementation of symmetric-key cryptographic algorithms (block/stream ciphers, hash functions, ...) looks attractive as well, thanks to the higher-order bitwise operations at the core of the current API.
Next, we shall investigate a full compilation process whose target will be machine code. Namely, we plan to go further beyond the first compilation phase of Section 4, where, in fact, we describe an in-line parameters unfolding of standard programming patterns like $\text{Map}[\cdot]$ and $\text{Fold}[\cdot, \cdot]$. The compilation to machine code will target parallelization, generally implied by functional programming thanks to its reduced data dependency.

Interestingly, while programming the binary field arithmetic, we found that the main programming patterns we used can be assimilated to the MapReduce paradigm [13]. This means that not only DLAL can be used to certify polynomial-time complexity, but it is also suitable to be adapted to actual cloud platforms based on the MapReduce.

Finally, we do not exclude that more refined logics than DLAL can be used to realize a similar framework with even better built-in properties. Our choice of DLAL originated as a trade-off between flexibility in programming and constrains imposed by the typing system, but it is at the same time an experiment. Different logics can for instance measure the space complexity, or provide a more fine-grained time complexity.

References


A Definition of Combinators

$b\text{Cast}^m$ is $\lambda b.(((b) 1) 0) \bot$.

$b\nabla_t$ is $\lambda b.(((b) (1 \ldots 1)) (0 \ldots 0)) (\bot \ldots \bot)$, for every $t \geq 2$. 

20
wSuc is $\lambda bp. \lambda f x. ((f) (b \text{Cast}^0) b) ((p) f) x$.

wCast$^m$ is $\lambda l. (((l) \varepsilon \text{Suc}^m) (wSuc) 0) \varepsilon \text{Suc}^m$, for every $m \geq 0$.

w$\nabla^t$ is $\lambda l. (((l) \varepsilon \text{Suc}^t) (w\text{Suc}^t) 0) \varepsilon \text{Suc}^t$, for every $t \geq 2$, and $m \geq 0$, where:

$$w\nabla^t \text{Step} \equiv \lambda b. \lambda (x_1 \ldots x_n). (((w\text{Suc}^t) b) x_1 \ldots (((w\text{Suc}^t) b) x_n)$$

$$w\nabla^t \text{Base} \equiv \langle \varepsilon \rangle \ldots \langle \varepsilon \rangle$$

Xor is: $\lambda bc. (((b) \lambda x. (((x) \varepsilon) 1) 1) \lambda x. (((x) \varepsilon) \varepsilon) \varepsilon) c$.

And is $\lambda bc. (((b) \lambda x. \lambda x. (((x) \varepsilon) \varepsilon) \varepsilon) \varepsilon) c$.

sSpl is $\lambda s. (s) \lambda l. (\langle \varepsilon, \ldots, \varepsilon \rangle) l x$.

wRev is $\lambda f x. (((l) \text{wRevStep} f) \lambda x. x) x$ where, if $f : \mathbb{B}_2 \rightarrow \alpha \rightarrow \alpha$, then $\text{wRevStep} f : \mathbb{B}_2 \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ is $\text{wRevStep} f \equiv \lambda e r x. (\langle (f) e \rangle) x$.

wDrop is $\lambda f x. (((l) \lambda e. (((e) \lambda f. (f) 1) \lambda f. (f) \varepsilon) \lambda f z z) f) x$.

w2s is $\lambda l. (((l) \lambda e s. (\langle e, s \rangle) (\varepsilon) \varepsilon) \varepsilon) l$.

Map$[F]$ is $\lambda f x. (((l) \lambda e. (((e) \lambda f. (F) e) \varepsilon) (\text{Cast}^0) (S)) \varepsilon), where F : A \rightarrow B must be a closed term.

Fold$[F, S]$ is $\lambda l. (((l) \lambda e z. (((e) \lambda f. (F) e) z) \lambda f z z) z) S$, where $F : A \rightarrow B \rightarrow B$ and $S : B \rightarrow \varepsilon$ must be closed terms.

MapThread$[F]$ is $\lambda m f x. (\lambda (w, s). w) (((l) \text{MTStep} [F, f]) \text{MTBase} [m, x])$, where $F : A \rightarrow B \rightarrow C must be a closed term; the step function $\text{MTStep} [F, f] : (\alpha \odot S) \rightarrow (\alpha \odot S)$ and the base function $\text{MTBase} [m, x] : \alpha \odot S$ are:

$$\text{MTStep} [F, f] \equiv \lambda a. \lambda (w, s). (\lambda (b, s'). ((f) (\langle F \rangle a) b) w, s')) (\text{sSpl} s)$$

$$\text{MTBase} [m, x] \equiv \langle x, (\text{w2s} (\text{wRev} m) \rangle$$

wProj is $\lambda f x. (((l) \lambda (a, b). \langle (f) a \rangle) b) x$.

MapState$[F]$ is $\lambda s f x. (\lambda (w, s'). w) (((l) \text{MSStep} [F, f]) \text{MSBase} [x]) (\text{Cast}^0) s$, where $F : (A \odot S) \rightarrow (B \odot S)$ must be a closed term; the step function $\text{MSStep} [F, f] : A \rightarrow (\alpha \odot S) \rightarrow (\alpha \odot S)$ and the base function $\text{MSBase} [x] : S \rightarrow (\alpha \odot S)$ are:

$$\text{MSStep} [F, f] \equiv \lambda e. \lambda (w, s). (\lambda (e', s'). ((f) (\langle F \rangle e') w, s')) (\langle F \rangle e, s)$$

$$\text{MSBase} [x] \equiv \lambda s. (x, s)$$

A.1 Proofs

$$\text{MapFromFold} [F] \equiv \text{Fold} [\lambda e p f x. (\langle (f) (F) e \rangle ((p) f) x, \varepsilon)]$$

$$\equiv \lambda l. (((l) \lambda e z. (((e) \lambda e p f x. (\langle (f) (F) e \rangle ((p) f) x, e) z) (\text{wCast}^0) \varepsilon) \downarrow. \lambda l. (((l) \lambda e p f x. (\langle (f) (F) e \rangle ((p) f) x, e) z) (\text{wCast}^0) \varepsilon) \downarrow \varepsilon$$

The base case is easy to check: $(\text{Map} [F]) \varepsilon \downarrow \varepsilon$ and $(\text{MapFromFold} [F]) \varepsilon \downarrow \varepsilon$.

We now prove the inductive case. Let $l \equiv \lambda f x. (((f) b_{n-1}) \ldots (((f) b_0) x$ be a Church word of length $n$. Assume that $(\text{Map} [F]) l \downarrow V and (\text{MapFromFold} [F]) l \downarrow V$. We want
to show that Map[f] and MapFromFold[f] reduce to the same term for a Church word 
\( l' \equiv \lambda f.((f) b) ((l) f) x \) of length \( n + 1 \).

\[
\text{(Map[f]) } l' \downarrow \boxed{\lambda f.x.((l') \lambda e.((f) (F) e) x)}
\]

\[
\lambda f.x.((\lambda f.((f) b) ((l) f) x) \lambda e.((f) (F) e) x)
\]

\[
\lambda f.x.((\lambda x.((\lambda e.((f) e) b) ((l) \lambda e.((f) (F) e) x) f)
\]

\[
\lambda f.x.((\lambda e.((f) e) b) ((\lambda s.((l) \lambda e.((f) (F) e) b) f) x)
\]

\[
\lambda f.x.((\lambda e.((f) e) b) b) ((V) f) x)
\]

\[
\text{ind. hyp. } \downarrow \lambda f.x.((f) b) ((V) f) x
\]

\[
\text{\textbf{(MapFromFold[f]) } } l' \downarrow \boxed{((l') \lambda e p f.((f) (F) e) ((p) f) x) [e]}
\]

\[
\lambda e p f.((f) ((l) f) x) \lambda e p f.((f) (F) e) ((p) f) x) [e]
\]

\[
\lambda x.((\lambda e p f.((f) (F) e) ((p) f) x) b) ((l) \lambda e p f.((f) (F) e) ((p) f) x) x) [e]
\]

\[
\downarrow \lambda e p f.((f) (F) e) ((p) f) x) b) ((l) \lambda e p f.((f) (F) e) ((p) f) x) x) [e]
\]

\[
\text{ind. hyp. } \downarrow ((\lambda e p f.((f) (F) e) ((p) f) x) b) V
\]

\[
\lambda p f.((f) (F) b) ((p) f) x) V
\]

\[
\downarrow \lambda f.((f) (F) b) ((V) f) x
\]
Size Analysis of Higher-Order Functions

Péter Diviánszky, Attila Góbi, and Tamás Kozsik

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary

Abstract. The paper describes a resource analysis method for strict functional languages. The method is applicable for higher order functions. It provides lower and upper bounds for strictly increasing resource costs like execution time and the number of total allocated memory during execution. The key component described here is a calculus which is used for abstract interpretation of expressions and functions with free variables. The result of the abstract interpretation is a system of mutually recursive equations for costs.

1 Introduction

This paper presents a static resource analysis method on programs written in a simple strict functional language with higher order functions. The method is based on abstract interpretation and yields lower and upper bounds for strictly increasing resource costs like execution time and the number of total allocated memory during execution. The paper does not consider how to solve the system of mutually recursive equations for costs, resulting from the abstract interpretation.

The paper develops not merely an algorithm, but a calculus in which reasoning about resource consumption of programs is possible. The calculus introduces an equivalence relation over expressions. Lower and upper bounds are achieved with the use of interval arithmetics.

Some limitations of the current state of the method are the following.

– We examine the cost of non-optimized code. A lot of code optimization technique make the execution faster and improve the memory use of the program by only constant factor. On the other hand, some optimization can drastically decrease the amount of the used resources, which should be examined in the future.
– Currently we examine those resources which are non-decreasing like the runtime and the sum of all memory allocated by the program during its run. For these resources we give estimated upper and lower bounds. (Another possibility would be the amortized resource analysis.)
– Currently we do not support user defined data structures, only a fixed set of built-in data structures (basic types, tuples, lists and the ‘sum’ data type).

* 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).
We suppose that the examined program will not abort. For example, the head function will not be applied on an empty list during the execution of the program.

2 Core Language

The language considered in this paper is a simple pure functional language with strict evaluation.

2.1 Lexical Grammar

Keywords: =, let, in, case, of, _, (,), \rightarrow, ,, ;, {, }.

The keywords ;, { and } will be hidden in the example codes; the indentation helps to infer them.

Other tokens of the core language:

Ident (variable) identifier begins with a lowercase letter and contains alphanumeric characters. Function names in expressions also belong to this category.

ConId A constructor identifier begins with an uppercase letter and contains alphanumeric characters. Furthermore, the following literals are also considered as constructors:

- Bounded integers (between $-2^{31}$ and $2^{31}-1$; stored on 4 bytes). Examples: 111, -22, 0.
- Double precision floating point numbers like 0.0, 0.13, -1e10.
- Unicode characters like 'a', 'b', '@', '\n'.

2.2 Syntax

The extended BNF syntax of the core language is as follows:

Program ::= Function*

Function ::= Ident -- function name
            Ident* -- variables
            "="
            Expression -- function body
            ";"

Expression ::= Ident -- variable or function name
             | ConId -- constructor or literal
             | Expression Expression+ -- application
             | "let" Ident "=" Expression "in" Expression -- let expr.
CaseAlt ::= Pattern "→" Expression ";

Pattern ::= "_" -- wildcard
| ConId Ident+ -- constructor name + variables

Restrictions. In case expressions only the last pattern may be a wildcard. In case expressions the first pattern may not be a wildcard.

Remarks. An expression can be enclosed by parentheses; this is not shown in the BNF syntax because we wanted to put emphasis on language constructs which influence resource analysis. Similarly some constructors and built-in functions have special syntax not shown in the BNF syntax above. See the next section for more information.

We don’t distinguish between “let” and “let rec”, because we allow only constant definitions which can not be recursive in a strictly evaluated language. The let construct introduces only one variable because let constructs with more variables can be expressed several simple let constructs (because we have no recursive constants).

A function definition may have multiple arguments because if it were translated into one argument function definitions then we could not distinguish between curried and saturated applications which is need for proper resource analysis. Similarly an application may have multiple arguments.

Pattern matching is only allowed in case expressions and nested pattern matching is not allowed. The reason is that nested pattern matching can be compiled to nested case expressions and this is what the compilers usually do.

2.3 Defined Entities

Built-in Data Constructors Here are the list of built-in data constructors. The arity of the constructors are given in parenthesis.

- Literals like integers, floating point numbers and characters (0).
- True, False (0).
- Unit (0)
- Tuple₂ (2), Tuple₃ (3), Tuple₄ (4), ...
  Tuples have special syntax: (Tuple₄ a b c d) is written as (a, b, c, d).
- Nil (0), Cons (2).
  Special syntax: Nil is written as [], (Cons a b) is written as (a:b).
- Left (1), Right (1).
Builtin Functions  The list of built-in (native) functions is as follows. We have to keep these functions in built-in status for proper resource calculation (a non-native addition would be slower). This list can be extended as needed.

- Operations on integers: \(+_{\text{int}}, -_{\text{int}}, \star_{\text{int}}, <_{\text{int}}, >_{\text{int}}, \leq_{\text{int}}, \geq_{\text{int}}, =_{\text{int}}, /=_{\text{int}}, \text{div}, \mod\)
- Operations on floating point numbers: similarly as needed.
- Operations on characters: as needed.

For simplicity, type indices will be omitted if they can be inferred from the environment. Infix notation will be used for functions whose name consists graphical characters. For example, \((1 + 2)\) stands instead of \((+_{\text{int}} 1 2)\).

Free Variables  Free variables in expressions are defined by the FV function as usual:

\[
\begin{align*}
\text{FV}(v) &= \{v\} \quad \text{— variable} \\
\text{FV}(f) &= \{\} \quad \text{— function name} \\
\text{FV}(C) &= \{\} \quad \text{— constructor} \\
\text{FV}(e_1 \ldots e_n) &= \text{application} \\
&= \text{FV}(e_1) \cup \ldots \cup \text{FV}(e_n) \\
\text{FV(let } v = e \text{ in } e) &= \text{FV}(e) \setminus \{v\} \\
\text{FV(case } e \text{ of } C_1 v_{11} \ldots -\rightarrow e_1; \ldots; C_n v_{n1} \ldots -\rightarrow e_n) &= \text{FV}(e) \cup \text{FV}(e_1) \setminus \{v_{11}, \ldots\} \cup \ldots \cup \text{FV}(e_n) \setminus \{v_{n1}, \ldots\}
\end{align*}
\]

The expression \(e\) is \textit{closed} if \(\text{FV}(e) = \emptyset\).

For any function definition \((f v_1 \ldots v_n = e)\) the following must hold:
\(\text{FV}(e) \subseteq \{v_1, \ldots, v_n\}\).

2.4 Arity

Every function and constructor has an arity which is a non-negative integer. The arity of a function is the number of variables in the left-hand-side of its definition.

We say that a function and constructor applications is \textit{saturated} if the number of arguments is equal to the arity of the function or constructor. In patterns, constructor applications should be saturated.

We say that a function and constructor application is \textit{curried} if the number of arguments is less than the arity of the function or constructor.

3 Equivalence of Expressions

First we define the equivalence of Core expressions which is needed for the definition of exact cost.\(^1\)

\(^1\) An alternative approach would be to define the reduction rules first, but then definition of the exact cost of constructor applications would be problematic.
(e₁ ∼ e₂) will denote that the expressions e₁ and e₂ are equivalent. We define the relation (∼) by the following rules.

The first rules are that (∼) is an equivalence relation, so it is reflexive, symmetric and transitive.

The second rule is the substitution property:
If e₁ ∼ e₂, then for any e expression and v variable e[v:=e₁] ∼ e[v:=e₂].

We will use the notation e[v:=e'] for variable substitution. It is computed by replacing all occurrence of v in e by e' (e and e' are expressions; v is a variable).

We will use the form e[v₁:=e₁, ..., vₙ:=eₙ] for simultaneous variable substitution for n variables.

Application of native functions are equivalent the corresponding results. For example, (1 + 2) ∼ 3 and (2 == 2) ∼ True.

Equivalence of other expressions are defined with the rules:

- (case C e₁ ... eₘ of p₁ → f₁; ...; pₙ → fₙ) ∼ fᵢ[v₁:=e₁, ..., vₘ:=eₘ] if (pᵢ = C v₁ ... vₘ) is the first matching pattern.
- (case C e₁ ... eₘ of p₁ → f₁; ...; pₙ → fₙ) ∼ fₙ if pₙ = _ and the previous rule is not applicable.
- (let v = e in e') ∼ e'[v:=e].
- (f e₁ ... eₙ) ∼ e[v₁:=e₁, ..., vₙ:=eₙ] if (f v₁ ... vₙ = e) is a function definition.
- ((e₁ ... eₙ) f₁ ... fₘ) ∼ (e₁ ... eₙ f₁ ... fₘ); this rule states that nested applications may be flattened.

Example: (let x = 1 + 3 in x * x) ∼ (8 + 8) because (let x = 1 + 3 in x * x) ∼ (let x = 4 in x * x) ∼ (4 * 4) ∼ 16 ∼ (8 + 8).

4 Exact Cost Calculation

4.1 Extended Non-negative Real Numbers

In some cases, the cost may be infinite (like the time consumption of an infinite cycle) so we will use the following set as the value set of cost:

\[ \mathbb{R}_\infty = [0, \infty] \] (a closed interval).

Basic operations and relations can be extended for \( \mathbb{R}_\infty \), like 3∞ = ∞ or 4 < ∞.

4.2 Marking Irrelevant Parts of Expressions

If a subexpression is irrelevant regarding cost calculation, we will prefix it with the (∼) sign like 4 + ∼ 5. Note that its always clear from the context whether (∼) denotes this mark or it is the previously defined equivalence relation.
4.3 Adjustable Cost Constants

During resource analysis, we will use several constants which are non-negative real numbers. These are denoted by Greek letters: \(\kappa_0, \kappa_1, \kappa_2, \ldots, \pi, \eta, \alpha_1, \alpha_2, \zeta_1, \zeta_2, \ldots \in [0, \infty[.\)

For example, \(\kappa_1\) will be the cost of creating a 1-arity constructor.

These constants can be tuned and adjusted to specific needs. For example, if one is interested in memory consumption only, the constants regarding reduction costs can be handled as 0. This is why we call these constants adjustable.\(^2\)

4.4 Denotation and General Rules

Let denote the (exact) cost of the closed expression \(e\) by \([e] \in \mathbb{R}_\infty\).

We define \([\cdot]\) by the following rules (first the general rules, then the language-construct-specific rules in the next subsections):

- If the expression is marked as irrelevant, then the cost is 0: \([\sim e] = 0\).
- Behind the (\(\sim\)) mark, any expression can be replaced by an equivalent one:
  \([e[v:= \sim e_1]] = [e[v:= \sim e_2]]\) if \(e_1 \sim e_2\).
- Chain rule: \([e[v:= f]] = [f] + [e[v:= \sim f]]\) if \(f\) is closed and the only occurrence of \(v\) in \(e\) is not behind a case expression (so it will be evaluated sooner or later).

Examples:
- \([\sim 42] = 0\).
- \([1 + \sim (1+1)] = [1 + \sim 2]\).
- \([1 + (2 + 3)] = [2 + 3] + [1 + \sim (2 + 3)] = [2 + 3] + [1 + \sim 5]\).

4.5 Saturated Constructor Applications

The cost of a saturated constructor application is \(\kappa_n\) where \(n\) is the arity of the constructor:

\([C \sim e_1 \ldots \sim e_n] = \kappa_n\).

\(\kappa_n\) can be explained as the cost of the creation of a constructor with \(n\) parameters in the memory.

Let \(\kappa = \kappa_0\). Examples:

- \([42] = \kappa\).
- \([[]] = \kappa\).
- \(['a'] = \kappa\).
- \([\text{True}] = \kappa\).
- \([\text{Just } 1] = \kappa_1 + \kappa\).
- \([\text{Just } \sim 1] = \kappa_1\).
- \([[]] = [[\sim 1 : []]] = [\text{Cons } \sim 1 []] = \kappa_2 + \kappa\).
- \([1, 2, 3] = [\sim 1 : \sim 2 : \sim 3 : []] = 3\kappa_2 + \kappa\).

\(^2\) Note that these adjustable constants could be eliminated in our theory if we would replace \(\mathbb{R}_\infty\), the set of cost values, by \(\mathbb{R}_\infty \times \ldots \times \mathbb{R}_\infty\) where the number of factors are equal the number of adjustable constants used here.
4.6 Native Function Applications

We assume that the cost of a native function application does not depend on its actual argument.

Let \( f \) a function with arity \( n \). Then
\[
[f \sim_e \ldots \sim_e] = |f|\delta.
\]
\( \delta \) is an adjustable constant. \(|\cdot|\) is a fixed function from native functions to \( \mathbb{R}_\infty \). A possible \(|\cdot|\) function is like
\[
|+|_{\text{int}} = 1, \\
|*|_{\text{int}} = 4, \\
|\text{mod}|_{\text{int}} = 16, \\
\ldots
\]
Examples:
- \([\sim 1 + \sim 2] = \delta \).
- \([\sim 1 + \sim 2 + \sim 3] = 2\delta \).
- \([\sim 3 \times \sim 15] = 4\delta \).

4.7 Case Expressions

Let \( \pi \) denote the cost of pattern matching. Then the cost of case distinction is:

- \( [\text{case } \sim(C f_1 \ldots f_m) \text{ of } p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n] = \pi + [e_1[v_1 := \sim f_1, \ldots, v_m := \sim f_m]] \)
  if \((p_i = C v_1 \ldots v_m)\) is the first matching pattern;
- \( [\text{case } \sim e \text{ of } p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n] = \pi + [e_n] \)
  if \( p_n = \_ \) and the previous rule is not applicable.

Example: \([\text{case } \sim[\_] \text{ of } \_ \rightarrow \text{True}; \_ \rightarrow \text{False}] = \pi + [\text{True}] = \pi + \kappa \)

4.8 Let Expressions

Let expression elimination has no additional cost:
\[
[\text{let } v = \sim e \text{ in } e'] = [e'][v := \sim e]
\]
Example: \([\text{let } x = 1 + 1 \text{ in } x + x] = [1 + 1] + [\text{let } x = \sim(1 + 1) \text{ in } x + x] = [1 + 1] + [\sim(1+1) + \sim(1+1)] = [1 + 1] + [\sim 2 + \sim 2] \).

4.9 Saturated Function Applications

Let \((f v_1 \ldots v_n = e)\) a function definition. The cost of a saturated function application is
\[
[f \sim_e \ldots \sim_e] = \alpha_n + [e[v_1 := \sim e_1, \ldots, v_n := \sim e_n]].
\]
For example, if \((\text{double } x = x + x)\) then \([\text{double } (1 + 1)] = [1 + 1] + \alpha_1 + [\sim(1+1) + \sim(1+1)] = 2\kappa + \delta + \alpha_1 + \delta \).
4.10 Currying

The cost of creating a curried application (a closure) with n additional arguments is $\zeta_m$. Examples:

- $[(\cdot)] = \zeta_0$
- $[[(+)] \sim 1] = \zeta_1$
- $[[f \sim 1] \sim 2] = \zeta_1 + \alpha_1$, if f has arity 2.
- $[[f \sim 1] \sim 2] = 2\zeta_1$, if f has arity 3.
- $[[f \sim 1] \sim 2] = \zeta_2$, if f has arity 3.
- $[[(+)] \sim 1] \sim 2] = \zeta_1 + \alpha_1 + \delta$

5 Extended Expressions

As shown in the introduction, abstract interpretation will be done on extended expressions.

Currently we extend the expressions with 2 new constructs:

1. (Arbitrary) $\tau$-typed expression: $\_^\tau$.
2. (Arbitrary) list with length n whose elements are $\tau$-typed: $\text{List}_n^\tau$.

We tend to omit the superscripts of these expressions.

5.1 Semantics of Extended Expressions

A (possibly not well-defined) semantics of the extended expressions can be given by a function which maps extended expressions to sets of normal expressions. Let denote this function by $\cdot$.

- $|e| = \{e\}$ if e is a constructor, variable or function symbol.
- $|\text{case } e \text{ of } p_1 \rightarrow f_1; \ldots; p_n \rightarrow f_n|$ = $\{\text{case } e' \text{ of } p_1 \rightarrow f_1'; \ldots; p_n \rightarrow f_n' \mid e' \in |e|, f_1' \in |f_1|, \ldots, f_n' \in |f_n|\}$.
- $|\text{let } v = e_1 \text{ in } e_2| = \{\text{let } v = e_1' \text{ in } e_2' \mid e_1' \in |e_1|, e_2' \in |e_2|\}$.
- $|e_1 \ldots e_n| = \{e_1' \ldots e_n' \mid e_1' \in |e_1|, \ldots, e_n' \in |e_n|\}$.
- $|\sim e| = \{\sim e' \mid e' \in |e|\}$
- $|\_^\tau| = \{\sim e' \mid e' \in \text{the set of all } \tau\text{-typed expression}\}$.
- $|\text{List}_0^\tau| = \{|\_|\}$.
- $|\text{List}_n^\tau| = |\sim(\_^\tau : \text{List}_n^\tau)|$.

3 This set is not well-defined; however this fact will not disturb our theory.
5.2 Equivalence of Extended Expressions

We can extend the equivalence of expressions to extended expressions: $e_1 \sim e_2$ if $\forall e_1' \in |e_1|, e_2' \in |e_2|: e_1' \sim e_2'$.

Examples:
- tail $\text{List}_{n+1} \tau \sim \text{List}_n \tau$
- head $\text{List}_{n+1} \tau \sim _\tau$
- reverse $\text{List}_n \tau \sim \text{List}_n \tau$

Note that the expression $(\text{filter } _\tau \rightarrow \text{Bool} \text{List}_n \tau)$ cannot be reduced. For this we would need the new construct “lists whose length are smaller than or equal to $n$”. If we introduced this, we still could find more abstract expressions which could not be reduced. This is exactly the situation which was said in the introduction: “One crucial requirement is to diminish precision so as to make problems manageable while still retaining enough precision for answering the important questions.”

6 Cost Estimation

6.1 Interval Arithmetic

Intervals are the value set of cost estimation so we have to now some basic operations on intervals.

Let $I = \{(a, b) \mid a, b \in \mathbb{R}_\infty, a \leq b\}$.

Operations on $I$:
- $(a, b) + (c, d) = (a+c, b+d)$.
- $n(a, b) = (na, nb)$.
- Union: $(a, b) \cup (c, d) = (\min a c, \max b d)$.
- Inclusion: $(a, b) \subseteq (c, d) \iff a \geq c \land b \leq d$.

We can say that the interval are the extension of $\mathbb{R}_\infty$, because a real number can be represented as a one-point interval. Sometimes we write $x$ instead of $(x, x)$, like in $1 + (2, 3)$ which is $(1, 1) + (2, 3)$.

6.2 Semantics of Cost Estimation

Cost estimation is the extension of the function $\lfloor \cdot \rfloor$ in such a way that the domain of the function is extended to extended expressions from expressions and the value set of the function is extended to intervals from $\mathbb{R}_\infty$.

Now we give the semantics of cost estimation.\footnote{Note that cost estimation is not well-defined definition because $|\cdot|$ is not well-defined.} Let $e$ be a closed extended expression.

$[e] = (a, b) \iff \forall e' \in |e|: a \leq [e'] \leq b$ and $(a, b)$ is the minimal such interval.

For an open extended expression $e$, the definition is:

$[e] = [e][v_1:= _\tau^1, \ldots, v_n:= _\tau^n]$

where $v_1, \ldots, v_n$ are the free variables of $e$ with types $\tau_1, \ldots, \tau_n$.\footnote{Note that cost estimation is not well-defined definition because $|\cdot|$ is not well-defined.}
7 Abstract Interpretation Rules

We defined the cost function, but the cost function is not computable. In this section we give some rules\(^5\) with which abstract interpretation can be performed:

- \([\sim e] = (0, 0) = 0\)
- \([e[v := \sim e_1]] = [e[v := \sim e_2]]\) if \(e_1 \sim e_2\).
- Chain rule: \([e[v := f]] = [f] + [e[v := \sim f]]\) if \(f\) is closed and the only occurrence of \(v\) in \(e\) is not behind a case expression (so it will be evaluated sooner or later).
- \([e[v := \sim e']] \subseteq [e[v := \tau]]\) if \(e'\) has type \(\tau\).
- \([v] = 0\) if \(v\) is a variable.
- \([C v_1 \ldots v_n] = \kappa_n\) if the application is saturated.
- \([f v_1 \ldots v_n] = \delta[f]\), if \(f\) is a native function and the application is saturated.
- \([\text{let } v = \sim e \text{ in } e'] = [e'[v := \sim e]]\).
- \([f \sim e_1 \ldots \sim e_n] = \alpha_n + [e[v_1 := e_1, \ldots, v_n := e_n]]\) if \((f v_1 \ldots v_n = e)\) is a function definition.
- \([\text{case } _\text{List}_0 \text{ of } [\vdash] \rightarrow e_1; v_1:v_2 \rightarrow e_2] = \pi + [e_1]\).
- \([\text{case } _\text{List}_{n+1} \text{ of } [\vdash] \rightarrow e_1; v_1:v_2 \rightarrow e_2] = \pi + [e_2[v_1 := _, v_2 := \sim \text{List}_n]]\).

7.1 Example

\[
\text{null } l = \text{case } l \text{ of } \\
[\vdash] \rightarrow \text{True} \\
_ \rightarrow \text{False} \\
\]

\[
[\text{null } _] = \alpha_1 + [\text{case } l \text{ of } \vdash \rightarrow \text{True}; _ \rightarrow \text{False}] = \alpha_1 + \pi + \kappa.
\]

7.2 Example

\[
\text{head } l = \text{case } l \text{ of } (x::_) \rightarrow x \\
\]

\[
[\text{head } _] = \alpha_1 + [\text{case } _ \text{ of } (x:_) \rightarrow x] = \alpha_1 + \pi.
\]

7.3 Example

\[
\text{a } \&\& \text{ b } = \text{case } a \text{ of } \\
\text{False } \rightarrow a \\
_ \rightarrow b \\
\]

\[
[\sim \&\& _] = \alpha_2 + [\text{case } _ \text{ of } \text{False } \rightarrow _; _ \rightarrow _] = \alpha_2 + \pi.
\]

\(^5\) These rules are in fact theorems but we omit the proofs this time.
7.4 Example

\[\text{and3 } a \ b \ c = a \land (b \land c)\]

\[\text{and3 } \_ \ _ \ _ = \alpha_3 + [\_ \land (\_ \land \_)] = \alpha_3 + 2(\alpha_2 + \pi).\]

7.5 Example

A second variant of \((\&\&)\):

\[a \land b = \text{case } a \text{ of }\]
\[\quad \text{False \to False}\]
\[\quad \_ \to b\]

\[\_ \land \_ = \alpha_2 + [\text{case } \_ \text{ of False \to False; } \_ \to \_] = \alpha_2 + \pi + (0, \kappa).\]

Note that this variant seems worse than the previous one because \(\alpha_2 + \pi \subseteq \alpha_2 + \pi + (0, \kappa)\). On the other hand, the compiler — if smart enough — produces equally fast code. This is because zero argument constructors like False are stored only in one place in memory. We can emulate this behavior by setting the constant \(\kappa\) to 0.

7.6 Example

\[\text{even } n = \mod n 2 == 0\]

\[\text{even } \_ = \alpha_1 + [\mod _2 == 0] = \alpha_1 + [\mod _2] + [\sim(\mod _2 == 0)] = \alpha_1 + 16\delta + \delta.\]

8 Driving Abstract Interpretation

Although abstract interpretation is mostly computable, some heuristics are needed to drive the interpretation process. We will show this by examples.

8.1 Example: length

Consider the following definition of list length.

\[\text{length } l = \text{case } l \text{ of }\]
\[\quad [] \to 0\]
\[\quad x:xs \to 1 + \text{length } xs\]

It would not be wise to try to calculate \([\text{length } \_]\). It would be \(\infty\), because computing the length of an arbitrary large list has arbitrary large cost. Instead of that, we calculate \([\text{length }\text{List}_0]\) and \([\text{length }\text{List}_{n+1}]\):

\[\text{length }\text{List}_0 = \alpha_1 + [\text{case }\text{List}_0 \text{ of } [] \to 0; x:xs \to 1 + \text{length } xs] = \]

33
\[ \alpha_1 + \pi + [0] = \]
\[ \alpha_1 + \pi + \kappa. \]

\[ [\text{length List}_{n+1}] = \]
\[ \alpha_1 + [\text{case List}_{n+1} \text{ of } [] \rightarrow 0; x:x \rightarrow 1 + \text{length } xs] = \]
\[ \alpha_1 + \pi + [1 + \text{length List}_n] = \]
\[ \alpha_1 + \pi + \kappa + \delta + [\text{length List}_n]. \]

During the calculation of \([\text{length List}_{n+1}]\) we stopped at \([\text{length List}_n]\) which is reasonable because we do not know whether \(n\) is zero or not.

Now let define an auxiliary function \(f(n) = [\text{length List}_n]\). Why we do this also need some heuristics but quite reasonable as we will see.

So far we have 2 equations:
\[ [\text{length List}_0] = \alpha_1 + \pi + \kappa, \]
\[ [\text{length List}_{n+1}] = \alpha_1 + \pi + \kappa + \delta + [\text{length List}_n], \]
which can be reformulated to an equation system for \(f\):
\[ f(0) = \alpha_1 + \pi + \kappa; \]
\[ f(n+1) = \alpha_1 + \pi + \kappa + \delta + f(n). \]

Suppose that somehow we know from this that \(f(n) = n(\alpha_1 + \pi + \kappa + \delta) + \alpha_1 + \pi + \kappa. \)

From this we get that
\[ [\text{length List}_n] = n(\alpha_1 + \pi + \kappa + \delta) + \alpha_1 + \pi + \kappa. \]
Note that this result is an exact solution, not an estimation, because this is a one-point interval for every \(n\).

### 8.2 Example: \text{length} (other version)

Consider the accumulating version of the \text{length} function:

\[ [\text{length List}] = f(0) \]
\[ f(i+1) = [\text{case List}_n \text{ of } [] \rightarrow i; x:x \rightarrow f(i+1) x] \]
\[ = f(_+1) \]
\[ \text{List}_n \]

Now it is wise to calculate \([f \ _ \ List_0]\) and \([f \ _ \ List_{n+1}]\):
\[ [f \ _ \ List_0] = \alpha_2 + \pi; \]
\[ [f \ _ \ List_{n+1}] = \alpha_2 + \pi + [f (_+1) \ List_n] = \alpha_2 + \pi + \kappa + \delta + [f \sim (_+1) \ List_n]. \]

The auxiliary function is \(g(n) = [f \ _ \ List_n]\). With this the equation system is:
\[ g(0) = \alpha_2 + \pi; \]
\[ g(n+1) = \alpha_2 + \pi + \kappa + \delta + g(n). \]

Finally we get that
\[ [f \ _ \ List_n] = n(\alpha_2 + \pi + \kappa + \delta) + \alpha_2 + \pi, \]
\[ [\text{length List}_n] = \alpha_1 + [f \ 0 \ List_n] = n(\alpha_2 + \pi + \kappa + \delta) + \alpha_1 + \alpha_2 + \pi + \kappa. \]

This is worse than previous version, although this is the more efficient version, which can be explained by not considering the tail-call optimization.

---

\[ ^6 \text{See Section ??} \]
\[ ^7 \text{Note that } [f \sim (_+1) \ List_n] = [f \ _ \ List_n]. \]
8.3 Example: reverse

The accumulating (more efficient) version of reverse:

\[
\text{reverse } l = \text{rev } [] \ l
\]

\[
\text{rev acc } l = \text{case } l \text{ of }
\]
\[
[] \rightarrow \text{acc}
\]
\[
x:xs \rightarrow \text{rev } (x:\text{acc}) \ xs
\]

The equations:
\[
[\text{rev } \text{List}_0] = \alpha_2 + \pi,
\]
\[
[\text{rev } \text{List}_{n+1}] = \alpha_2 + \pi + \kappa_2 + [\text{rev } \text{List}_n].
\]

From this we get that
\[
[\text{rev } \text{List}_n] = \alpha_1 + [\text{rev } \text{List}_n] = n(\alpha_2 + \pi + \kappa_2) + \alpha_2 + \alpha_1 + \pi + \kappa.
\]

8.4 Example: List Concatenation

We need this function for the next example:

\[
l ++ m = \text{case } l \text{ of }
\]
\[
[] \rightarrow m
\]
\[
x:xs \rightarrow x: (xs ++ m)
\]

The equations:
\[
[\text{List}_0 ++ \_] = \alpha_2 + \pi;
\]
\[
[\text{List}_{n+1} ++ \_] = \alpha_2 + \pi + \kappa_2 + [\text{List}_n ++ \_].
\]

The result:
\[
[\text{List}_n ++ \_] = n(\alpha_2 + \pi + \kappa_2) + \alpha_2 + \pi.
\]

8.5 Example: reverse (naive version)

Consider the naive version of reverse:

\[
\text{reverse } l = \text{case } l \text{ of }
\]
\[
[] \rightarrow l
\]
\[
(x:xs) \rightarrow \text{reverse } xs ++ [x]
\]

The equations:
\[
[\text{reverse } \text{List}_0] = \alpha_1 + \pi;
\]
\[
[\text{reverse } \text{List}_{n+1}] = \alpha_2 + \pi + [\text{reverse } \text{List}_n ++ [\_]] = \alpha_2 + \pi + [\sim(\text{reverse } \text{List}_n) ++ \sim[\_]] + [\text{reverse } \text{List}_n] =^8 \alpha_2 + \pi + [\text{List}_n ++ \sim[\_]] + [\text{reverse } \text{List}_n] =}
\]

\[^8\text{Note that in this step we used the information that (reverse } \text{List}_n = \text{List}_n).\]
\[\alpha_2 + \pi + \kappa_2 + \kappa + \text{List}_n ++ \sim \text{List}_n + \text{reverse List}_n = \]
\[\alpha_2 + \pi + \kappa_2 + \kappa + n(\alpha_2 + \pi + \kappa_2) + \alpha_2 + \pi + \text{reverse List}_n = \]
\[2\alpha_2 + 2\pi + \kappa_2 + \kappa + n(\alpha_2 + \pi + \kappa_2) + \text{reverse List}_n.\]

The solution:
\[\text{reverse List}_n = n(2\alpha_2 + 2\pi + \kappa_2 + \kappa + n(\alpha_2 + \pi + \kappa_2)) + \alpha_1 + \pi.\]
From this we can see that the naive reverse algorithm is \(O(n^2)\).

### 8.6 Example: filter

Let us see a slightly more complicated example, with higher order functions.

```haskell
filter p l = case l of
  [] → 1
  x:xs → case p x of
    True → x: filter p xs
    _ → filter p xs
```

First compute the usual equations. Note that we compute \([\text{filter p List}_{n+1}]\) instead of \([\text{filter } _ \text{List}_{n+1}]\) because we would like to see \(p\) in the result (it will not disappear). In general, a unique variable name may be used instead of every \((\_\_\_)\) expression.

\[[\text{filter p List}_0] = \alpha_2 + [\text{case List}_0 \text{ of } [\rightarrow 1]; x:xs \rightarrow \text{case } p \text{ x of True } \rightarrow x: \text{filter p xs}; _ \rightarrow \text{filter p xs}] = \]
\[\alpha_2 + \pi + [1] = \]
\[\alpha_2 + \pi + \kappa; \]
\[[\text{filter p List}_{n+1}] = \alpha_2 + [\text{case List}_{n+1} \text{ of } [\rightarrow 1]; x:xs \rightarrow \text{case } p \text{ x of True } \rightarrow x: \text{filter p xs}; _ \rightarrow \text{filter p xs}] = \]
\[\alpha_2 + \pi + [\text{case } p \_ \text{ of True } \rightarrow x: \text{filter p List}_n; _ \rightarrow \text{filter p List}_n] = \]
\[\alpha_2 + \pi + [p \_ \_ + \pi + [\_ : \text{filter p List}_n] || [\text{filter p List}_n] = \]
\[\alpha_2 + \pi + [p \_ \_ + \pi + (\kappa_2 + [\text{filter p List}_n]) || [\text{filter p List}_n] = \]
\[\alpha_2 + \pi + [p \_ \_ + \pi + [\text{filter p List}_n] + (0, \kappa_2) = \]
\[\alpha_2 + 2\pi + [p \_ \_ + [\text{filter p List}_n] + (0, \kappa_2).\]

The auxiliary function: \(f_p(n) = [\text{filter p List}_n].\)

The equation system:
\[f_p(0) = \alpha_2 + \pi + \kappa\]
\[f_p(n+1) = \alpha_2 + [p \_ \_] + 2\pi + (0, \kappa_2) + f_p(n).\]

The solution:
\[f_p(n) = n(\alpha_2 + [p \_ \_] + 2\pi + \kappa_2) + \alpha_2 + \pi + \kappa, \text{ so}\]
\[[\text{filter p List}_n] = n(\alpha_2 + [p \_ \_] + 2\pi + \kappa_2) + \alpha_2 + \pi + \kappa.\]

Let us see an application:
\[[\text{filter even } \sim [1..10]] \subseteq \]
\[[\text{filter even List}_{10}] = \]
\[10(\alpha_2 + [\text{even } \_ \_] + 2\pi + (0, \kappa_2)) + \alpha_2 + \pi + \kappa = 36\]
\[10(\alpha_2 + \alpha_1 + 17\delta + 2\pi + (0, \kappa_2)) + \alpha_2 + \pi + \kappa =
11\alpha_2 + 10\alpha_1 + 170\delta + 21\pi + \kappa + (0, 10\kappa_2).\]

This means that \([\text{filter even} \sim [1..10]]\) is between
\[11\alpha_2 + 10\alpha_1 + 170\delta + 21\pi + \kappa\]
and
\[11\alpha_2 + 10\alpha_1 + 170\delta + 21\pi + \kappa + 10\kappa_2.\]

The exact value is \(11\alpha_2 + 10\alpha_1 + 170\delta + 21\pi + \kappa + 5\kappa_2\), so the estimation is quite good.

9 Related work
TODO.

10 Future plans

- Amortized resource analysis.
- Improve the model to make better estimation for non-uniform data structures, like lists of lists with different size.

11 Conclusions
TODO.

References
Interval-based Resource Usage Verification: 
Formalization and Prototype

Pedro Lopez-Garcia\textsuperscript{1,2}, Luthfi Darmawan\textsuperscript{1}, Francisco Bueno\textsuperscript{3}, and Manuel Hermenegildo\textsuperscript{1,3}

\textsuperscript{1} The IMDEA Software Institute, Madrid, Spain
\textsuperscript{2} Spanish National Research Council (CSIC), Spain
\textsuperscript{3} Technical University of Madrid, Spain

\{pedro.lopez,luthfi.darmawan,manuel.hermenegildo\}@imdea.org
bueno@fi.upm.es

Abstract. In an increasing number of applications (e.g., in embedded, real-time, or mobile systems) it is important or even essential to ensure conformance with respect to a specification expressing resource usages, such as execution time, memory, energy, or user-defined resources. In previous work we have presented a novel framework for data size-aware, static resource usage verification. Specifications can include both lower and upper bound resource usage functions. In order to statically check such specifications, both upper- and lower-bound resource usage functions (on input data sizes) approximating the actual resource usage of the program which are automatically inferred and compared against the specification. The outcome of the static checking of assertions can express intervals for the input data sizes such that a given specification can be proved for some intervals but disproved for others. After an overview of the approach in this paper we provide a number of novel contributions: we present a full formalization, and we report on and provide results from an implementation within the Ciao/CiaoPP framework (which provides a general, unified platform for static and run-time verification, as well as unit testing). We also generalize the checking of assertions to allow preconditions expressing intervals within which the input data size of a program is supposed to lie (i.e., intervals for which each assertion is applicable), and we extend the class of resource usage functions that can be checked.

Key words: Cost Analysis, Resource Usage Analysis, Resource Usage Verification, Program Verification and Debugging.

1 Introduction and Motivation

The conventional understanding of software correctness is the conformance to a functional or behavioral specification, i.e., with respect to what the program is supposed to compute or do. However, in an increasing number of applications, particularly those running on devices with limited resources, it is also important
and sometimes essential to ensure the conformance with respect to specifications
expressing resource usages (such as execution time, memory, energy, or user-de-
defined resources). For example, in a real-time application, a program completing
an action later than expected is as erroneous as a program not computing the
correct answer. The same applies to an embedded application in a battery-oper-
ated device (for example in the medical or mobile phone domains) which makes
the device to run-out of batteries earlier than expected, and thus, making the
whole system useless.

In [12] we proposed techniques that extended the capacity of debugging and
verification systems based on static analysis [4, 2, 10] when dealing with a quite
general class of properties related to resource usage. This includes upper and
lower bounds on execution time, memory, energy, and user-defined resources
(the later in the sense of [15]). Such bounds are given as functions on input data
sizes (see [15] for the different metrics that can be used to measure data sizes,
such as list-length, term-depth or term-size). For example, the techniques of [12]
extend the capacity of CiaoPP to certify programs with resource consumption
assurances and also to efficiently check such certificates. We also defined an
abstract semantics for resource usage properties and described operations to
compare the (approximated) intended semantics of a program (i.e., the specifi-
cation, given as assertions in the program) with approximated semantics inferred
by static analysis, all for the case of resources. Thus, these operations include
the comparison of arithmetic functions (in particular, for [12], polynomial and
exponential functions).

In traditional static checking-based verification (e.g., [4]), for each property or
(part of) an assertion, the possible outcomes are true (property proved to hold),
false (property proved not to hold), and unknown (the analysis cannot prove
true or false). However, it is very common that cost functions have intersections,
so that for a given interval of input data sizes, one of them is smaller than the
other one, but for another interval it is the other way around. Thus, a novel
aspect of the resource verification and debugging approach proposed in [12] is
that the answers of the checking process go beyond these classical outcomes and
typically include conditions under which the truth or falsity of the property can
be proved. Such conditions can be parameterized by attributes of inputs, such
as input data size or value ranges. For example, it may be possible to say that
the outcome is true if the input data size is in a given range and false if it is in
another one.

Consider for example the naive reverse program in Figure 1, with the classical
definition of predicate append. The assertion:

:- check_comp nrev(A,B)
   + (cost(lb, steps, length(A)), cost(ub, steps, 10*length(A))).

is a resource usage specification to be checked by CiaoPP. It uses the cost/3
property for expressing a resource usage as a function on input data sizes (third
argument) for a particular resource (second argument), approximated in the
way expressed by the first argument (e.g., lb for lower bounds and ub for up-
per bounds). The assertion expresses both an upper and a lower bound for
module(rev, [nrev/2], [assertions,regtypes,nativeprops,predefres(res_steps)]).

entry nrev(A,B) : (list(A, gnd), var(B)).
check comp nrev(A,B)
    + (cost(lb, steps, length(A)), cost(ub, steps, 10*length(A))).

nrev([],[]).
nrev([H|L],R) :- nrev(L,R1), append(R1,[H],R).

Fig. 1. A module for naive reverse.

Fig. 2. Resource usage functions for program naive reverse.

the number of resolution steps performed by \texttt{nrev(A,B)}, given as functions on the length of the input list \texttt{A}. In other words, it specifies that the resource usage (given in number of resolution steps) of \texttt{nrev(A,B)} lies in the interval \([\text{length}(A), 10 \times \text{length}(A)]\).

Each Ciao assertion can be in a verification status, marked by prefixing the assertion itself with keywords such as \texttt{check}, \texttt{checked}, \texttt{false}, or \texttt{true}. This specifies respectively whether the assertion is provided by the programmer and is to be checked, or is the result of processing an input assertion and proving it correct or false, or is the output of static analysis and thus correct (safely approximated) information. Omitting this prefix means “to be checked” [16].

The outcome of the static checking of the previous assertion is the following set of assertions:

\begin{verbatim}
:- false comp nrev(A,-1)
    : intervals(length(A),[i(0,0),i(17, inf)])
    + ( cost(lb,steps,length(A)), cost(ub,steps,10*length(A)) ).

:- checked comp nrev(A,-1)
    : intervals(length(A),[i(1,16)])
    + ( cost(lb,steps,length(A)), cost(ub,steps,10*length(A)) ).
\end{verbatim}
meaning that the assertion is false for values of \( length(A) \) belonging to the interval \([0, 0] \cup [17, \infty] \), and true for values of \( length(A) \) in the interval \([1, 16] \). In order to produce that outcome, CiaoPP’s resource analysis infers both upper and lower bounds for the number of resolution steps of \( nrev/2 \), which are compared against the specification. In this particular case, the upper and lower bound inferred by the analysis are the same, namely the function \( 0.5 \times length(A)^2 + 1.5 \times length(A) + 1 \) (which means that it is the exact resource usage function for \( nrev/2 \)).

As we can see in Figure 2, the resource usage function inferred by CiaoPP lies in the resource usage interval expressed by the specification, namely \([length(A), 10 \times length(A)]\), for \( length(A) \) belonging to the data size interval \([1, 16] \). Therefore, CiaoPP says that the assertion is checked in that data size interval. However for \( length(A) = 0 \) or \( length(A) \in [17, \infty] \), the assertion is false. This is because the resource usage interval inferred by the analysis is disjoint with the one expressed in the specification. This is determined by the fact that the lower bound resource usage function inferred by the analysis is greater that the upper bound resource usage function expressed in the specification.

In this paper we extend our previous work [12] in several ways: (a) presenting a complete formalization of the resource usage verification framework, and (b) reporting on a prototype implementation and experimental results. We also (c) extend the framework to deal with specifications containing assertions that include preconditions expressing intervals, and (d) extend the class of resource usage functions that can be checked (e.g., summatory and logarithm functions).

In order to illustrate (c) above, consider that often in a system the possible input data belong to certain value ranges. We extend the model to make it possible to express specifications whose applicability is restricted to intervals of input data sizes (previously this capability was limited to the output of the analyzer). This is useful to reduce false negative errors during static checking which may be caused by input values that actually never occur. To this end (and also to allow the system to express inferred properties in a better way w.r.t. [12]) we have extended the Ciao assertion language with the new intervals/2 property, for expressing such preconditions, used already in the previous output. Consider the previous example, and assume now that the possible length of the input list to be reversed is in interval \([1, 10]\). In this case, we can add a precondition to the specification expressing an interval for the input data size as follows:

\[
:- \text{check comp nrev(A,B)} : \text{intervals(length(A),[i(1,10)])} + (\text{cost(lb, steps, length(A))}, \text{cost(ub, steps, 10*length(A)))}).
\]

As we can see in Figure 2, this assertion is true, because for input values \( A \) such that \( length(A) \in [1, 10] \), the resource usage function of the program inferred by analysis lies in the specified resource usage interval \([length(A), 10 \times length(A)]\).

In general, the outcome of the static checking of an assertion with a precondition expressing an interval for the input data size can be different for different subintervals of the one expressed in the precondition.

The closest related work we are aware of presents a method for comparison of cost functions inferred by the COSTA system for Java bytecode [1]. The
method proves whether a cost function is smaller than another one for all the values of a given initial set of input data sizes. The result of this comparison is a boolean value. However, as mentioned before, in our approach the result is in general a set of subsets (intervals) in which the initial set of input data sizes is partitioned, so that the result of the comparison is different for each subset. The method in [1] also differs from the one presented here in that comparison is syntactic, using a method similar to what was already being done in the CiaoPP system: performing a function normalization and then using some syntactic comparison rules. However, in this work we go beyond these syntactic comparison rules. Moreover, we present an application for which cost function comparison is instrumental and which is not covered in the cited work: verification of resource usage properties. This implies extending the criteria of correctness and defining a resource usage (abstract) semantics and conditions under which a program is correct or incorrect with respect to an (approximated) intended semantics.

The structure of the rest of the paper is the following: Section 2 recalls the CiaoPP verification framework and Section 3 describes how it is used and extended for the verification of general resource usage program properties, and presents the formalization of the framework. Section 4 then briefly explains the technique that we have developed for resource usage function comparison. Section 5 reports on the implementation of our techniques within the Ciao/CiaoPP system, providing experimental results, and finally Section 6 summarizes our conclusions.

2 Foundations of the Verification Framework

Our work on data size-aware, static resource usage verification presented in [12] and in this paper builds on top of the previously existing framework for static verification and debugging [17], which is implemented and integrated in the CiaoPP system [10]. Our initial work on resource usage verification reported, e.g., in [10] and previous papers, was based on a different type of cost function comparison, basically consisting on performing function normalization and then using some syntactic comparison rules. Also, the outcome of the assertion checking was the classical one (true, false, or unknown), and did not produce intervals of input data sizes for which the verification result is different.

The verification and debugging framework of CiaoPP uses abstract interpretation-based analyses, which are provably correct and also practical, in order to statically compute semantic approximations of programs. These semantic approximations are compared with (partial) specifications, in the form of assertions that are written by the programmer, in order to detect inconsistencies or to prove such assertions.

Both program verification and debugging compare the actual semantics $[P]$ of a program $P$ with an intended semantics $I$ for the same program, which we will denote by $I$. This intended semantics embodies the user’s requirements, i.e., it is an expression of the user’s expectations. In Table 1 we show classical verification problems in a set-theoretic formulation as simple relations between $[P]$ and $I$. Using the exact actual or intended semantics for automatic verification and
debugging is in general not realistic, since the exact semantics can be typically only partially known, infinite, too expensive to compute, etc. On the other hand the technique of abstract interpretation allows computing safe approximations of the program semantics. The key idea of the CiaoPP approach is to use the abstract approximation \([P]_\alpha\) directly in program verification and debugging tasks.

A number of other approaches have also been proposed which make use to some extent of abstract interpretation in verification and/or debugging tasks. For example, abstractions were used in the context of algorithmic debugging in [11]. Abstract interpretation (generally for debugging of imperative programs) was studied by Bourdoncle [3], by Comini et al. for the particular case of algorithmic debugging of logic programs [6] (making use of partial specifications) [5], by P. Cousot [7], and others. Additional discussion and more details about the foundations and implementation issues of the CiaoPP approach can be found in [4, 9, 10].

**Abstract Verification and Debugging** In the CiaoPP framework the abstract approximation \([P]_\alpha\) of the concrete semantics \([P]\) of the program is actually computed and compared directly to the (also approximate) intention (which is given in terms of assertions [16]), following almost directly the scheme of Table 1. We safely assume that the program specification is given as an abstract value \(I_\alpha \in D_\alpha\) (where \(D_\alpha\) is the abstract domain of computation). Program verification is then performed by comparing \(I_\alpha\) and \([P]_\alpha\). Table 2 shows sufficient conditions for correctness and completeness w.r.t. \(I_\alpha\), which can be used when \([P]\) is approximated. Several instrumental conclusions can be drawn from these relations.

Analyses which over-approximate the actual semantics (i.e., those denoted as \([P]_{\alpha+}\)), are specially suited for proving partial correctness and incompleteness with respect to the abstract specification \(I_\alpha\). It will also be sometimes possible to prove incorrectness in the case in which the semantics inferred for the program is incompatible with the abstract specification, i.e., when \([P]_{\alpha+} \cap I_\alpha = \emptyset\). On the other hand, we use \([P]_{\alpha-}\) to denote the (less frequent) case in which analysis under-approximates the actual semantics. In such case, it will be possible to prove completeness and incorrectness.

Since most of the properties being inferred are in general undecidable at compile-time, the inference technique used, abstract interpretation, is necessarily

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P) is partially correct w.r.t.</td>
<td>([P] \subseteq I)</td>
</tr>
<tr>
<td>(P) is complete w.r.t.</td>
<td>(I \subseteq [P])</td>
</tr>
<tr>
<td>(P) is incorrect w.r.t.</td>
<td>([P] \not\subseteq I)</td>
</tr>
<tr>
<td>(P) is incomplete w.r.t.</td>
<td>(I \not\subseteq [P])</td>
</tr>
</tbody>
</table>

**Table 1.** Set theoretic formulation of verification problems
approximate, i.e., possibly imprecise. Nevertheless, such approximations are also always guaranteed to be safe, in the sense that they are never incorrect.

3 Extending the Framework to Data Size-Aware Resource Usage Verification

As mentioned before, our data size-aware resource usage verification framework is characterized by being able to deal with specifications that include both lower and upper bound resource usage functions (i.e., specifications that express intervals where the resource usage is supposed to be included in), and, in an extension of [12], that include preconditions expressing intervals within which the input data size of a program is supposed to lie. We start by providing a more complete formalization than that of [12] in order to define all the elements of the CiaoPP framework for its application to data size-aware resource usage verification.

3.1 Resource usage semantics

Given a program \( p \), let \( C_p \) be the set of all calls to \( p \). The concrete resource usage semantics of a program \( p \), for a particular resource of interest, \( \llbracket P \rrbracket \), is a set of pairs \((p(\overline{t}), r)\) such that \( \overline{t} \) is a tuple of terms, \( p(\overline{t}) \in C_p \) is a call to predicate \( p \) with actual parameters \( \overline{t} \), and \( r \) is a number expressing the amount of resource usage of the computation of the call \( p(\overline{t}) \). Such a semantic object can be computed by a suitable operational semantics, such as SLD-resolution, adorned with the computation of the resource usage. We abstract away such computation, since it will in general be dependent on the particular resource \( r \) refers to. The concrete resource usage semantics can be defined as a function \( \llbracket P \rrbracket : C_p \rightarrow \mathcal{R} \) where \( \mathcal{R} \) is the set of real numbers (note that depending on the type of resource we can take another set of numbers, e.g., the set of natural numbers).

The abstract resource usage semantics is a set of 4-tuples:

\((p(\overline{v}) : c(\overline{v}), \Phi, input_p, size_p)\)

where \( p(\overline{v}) : c(\overline{v}) \), is an abstraction of a set of calls. \( \overline{v} \) is a tuple of variables and \( c(\overline{v}) \) is an abstraction representing a set of tuples of terms which are instances of \( \overline{v} \). \( c(\overline{v}) \) is an element of some abstract domain expressing instantiation states. \( \Phi \), is an abstraction of the resource usage of the calls represented by \( p(\overline{v}) : c(\overline{v}) \).

We refer to it as a resource usage interval function for \( p \), defined as follows:

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>P is partially correct w.r.t. ( I_\alpha )</td>
<td>( \alpha(\llbracket P \rrbracket) \subseteq I_\alpha )</td>
<td>( [P]<em>{\alpha+} \subseteq I</em>\alpha )</td>
</tr>
<tr>
<td>P is complete w.r.t. ( I_\alpha )</td>
<td>( I_\alpha \subseteq \alpha(\llbracket P \rrbracket) )</td>
<td>( I_\alpha \subseteq [P]_{\alpha-} )</td>
</tr>
<tr>
<td>P is incorrect w.r.t. ( I_\alpha )</td>
<td>( \alpha(\llbracket P \rrbracket) \not\subseteq I_\alpha )</td>
<td>( [P]<em>{\alpha+} \not\subseteq I</em>\alpha ), or ( [P]<em>{\alpha+} \cap I</em>\alpha = \emptyset \land [P]_{\alpha} \not\neq \emptyset )</td>
</tr>
<tr>
<td>P is incomplete w.r.t. ( I_\alpha )</td>
<td>( I_\alpha \not\subseteq \alpha(\llbracket P \rrbracket) )</td>
<td>( I_\alpha \not\subseteq [P]_{\alpha+} )</td>
</tr>
</tbody>
</table>

Table 2. Verification problems using approximations
Definition 1. A resource usage bound function for $p$ is a monotonic arithmetic function, $\Psi : S \mapsto \mathbb{R}_\infty$, for a given subset $S \subseteq \mathbb{R}^k$, where $\mathbb{R}$ is the set of real numbers, $k$ is the number of input arguments to predicate $p$, and $\mathbb{R}_\infty$ is the set of real numbers augmented with the special symbols $\infty$ and $-\infty$. We use such functions to express lower and upper bounds on the resource usage of predicate $p$ depending on input data sizes.

Definition 2. A resource usage interval function for $p$ is an arithmetic function, $\Phi : S \mapsto \mathbb{RI}$, where $S$ is defined as before and $\mathbb{RI}$ is the set of intervals of real numbers, such that $\Phi(\bar{\vec{n}}) = [\Phi^l(\bar{\vec{n}}), \Phi^u(\bar{\vec{n}})]$ for all $\bar{\vec{n}} \in S$, where $\Phi^l(\bar{\vec{n}})$ and $\Phi^u(\bar{\vec{n}})$ are resource usage bound functions that denote the lower and upper endpoints of the interval $\Phi(\bar{\vec{n}})$ respectively for the tuple of input data sizes $\bar{\vec{n}}$. Although $\bar{\vec{n}}$ is typically a tuple of natural numbers, we do not want to restrict our framework. We require that $\Phi$ be well defined so that $\forall \bar{\vec{n}} (\Phi^l(\bar{\vec{n}}) \leq \Phi^u(\bar{\vec{n})))$.

`input_p` is a function that takes a tuple of terms $\bar{\vec{t}}$ and returns a tuple with the input arguments to $p$. This function can be inferred by using existing mode analysis or can be given by the user by means of assertions. `size_p(\bar{\vec{t}})` is a function that takes a tuple of terms $\bar{\vec{t}}$ and returns a tuple with the sizes of those terms under a given metric. The metric used for measuring the size of each argument of $p$ can be automatically inferred (based on type analysis information) or can be given by the user by means of assertions [15].

Example 1. Consider for example the naive reverse program in Figure 1, with the classical definition of predicate `append`. The first argument of `nrev` is declared input, and the two first arguments of `append` are consequently inferred to be also input. The size measure for all of them is inferred to be `list-length`. Then, we have that:

- `input_{nrev}(x,y) = (x)`, `input_{app}(x,y,z) = (x,y)`,
- `size_{nrev}(x) = (\text{length}(x))` and `size_{app}(x,y) = (\text{length}(x),\text{length}(y))`.

In order to make the presentation simpler, we will omit the `input_p` and `size_p` functions in abstract tuples, with the understanding that they are present in all such tuples.

Intended meaning The intended approximate meaning $I_{\alpha}$ of a program is an abstract semantic object with the same kind of tuples: $(p(\vec{v}) : \alpha(\vec{v}), \Phi, \text{input}_p, \text{size}_p)$, which are given in the form of assertions. The basic form of a resource usage assertion is:

$$\text{:- comp Pred [: Precond ] + ResUsage.}$$

which expresses that for any call to `Pred`, if `Precond` is satisfied in the calling state, then `ResUsage` should also be satisfied for the computation of `Pred`. `ResUsage` defines in general an interval of numbers for the particular resource usage of the computation of the call to `Pred` (i.e., `ResUsage` is satisfied by the computation of the call to `Pred` if the resource usage of such computation is in the defined interval).
Example 2. In the program of Figure 1 one could use the assertion:

```
:- comp nrev(A,B): ( list(A, gnd), var(B) )
+ resource(ub, steps, 1+exp(length(A), 2)).
```

to express that for any call to `nrev(A,B)` with the first argument bound to a ground list and the second one a free variable, an upper bound (ub) on the number of resolution steps performed by the computation is $1 + n^2$, where $n = \text{length}(A)$. In this case, the interval approximating the number of resolution steps is $[0, 1 + n^2]$. Since the number of resolution steps cannot be negative, the minimum of the interval is zero. If we assume that the resource usage can be negative, the interval would be $(-\infty, 1 + n^2]$. If we had a lower bound (lb) instead of an upper bound in the assertion, the interval would be $[1 + n^2, \infty)$. Such an assertion describes a tuple in $I_\alpha$ which is given by

$(p(\bar{v}) : c(\bar{v}), \Phi, \text{input}_p, \text{size}_p)$,

where $p(\bar{v}) : c(\bar{v})$ is defined by Pred and Precond, and $\Phi$ is defined by ResUsage. For simplicity, we assume that Pred is actually $p(\bar{v})$ and that there is a syntactic correspondence from Precond to $c(\bar{v})$, and from ResUsage to $\Phi$. The information about input$_p$ and size$_p$ is implicit in ResUsage.

The concretization of $I_\alpha$, $\gamma(I_\alpha)$, is the set of all pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of terms and $p(\bar{t})$ is an instance of Pred that meets precondition Precond, and $r$ is a number that meets the condition expressed by ResUsage (i.e., $r$ lies in the interval defined by ResUsage) for some assertion.

Example 3. The assertion in Example 2 captures the following concrete semantic tuples:

- $(\text{nrev}([a,b,c,d,e,f,g],X), 35)$
- $(\text{nrev}([],Y), 1)$

but it does not capture the following ones:

- $(\text{nrev}([A,B,C,D,E,F,G],X), 35)$
- $(\text{nrev}(W,Y), 1)$
- $(\text{nrev}([a,b,c,d,e,f,g],X), 53)$
- $(\text{nrev}([],Y), 11)$

Those in the first line above because they correspond to calls which are outside the scope of the assertion (i.e., they do not meet the precondition Precond); those on the second line (which will never occur during execution) because they violate the assertion (i.e., they meet the precondition Precond, but do not meet the condition expressed by ResUsage).

Partial correctness: comparing the abstract semantics.

Definition 3. Given a program $p$ and an intended resource usage semantics $I$, where $I : C_p \rightarrow R$, we say that $p$ is partially correct w.r.t. $I$ if for all $p(\bar{t}) \in C_p$ such that $r$ is the amount of resource usage of the computation of the call $p(\bar{t})$, we have that $I(p(\bar{t})) = r$, i.e., $(p(\bar{t}), r) \in I$. This is equivalent to the condition $\text{[[P]]} \subseteq I$ given in Table 1.

Definition 4. We say that $p$ is partially correct with respect to a tuple of the form $(p(\bar{v}) : c_1(\bar{v}), \Phi_1)$ if for all $p(\bar{t}) \in C_p$ such that $r$ is the amount of resource usage of the computation of the call $p(\bar{t})$, it holds that: if $p(\bar{t}) \in \gamma(p(\bar{v}) : c_1(\bar{v}))$ then $r \in \Phi_1(\bar{s})$, where $\bar{s} = \text{size}_p(\text{input}_p(p(\bar{t})))$. 

46
Definition 5. Given an intended abstract resource usage semantics $I_\alpha$ expressed as a set of tuples of the form $(p(\bar{v}) : c_1(\bar{v}), \Phi_1)$ (each tuple is expressed by an assertion in the program), we say that $p$ is partially correct with respect to $I_\alpha$ if for all $p(\bar{t}) \in C_p$ such that $r$ is the amount of resource usage of the computation of the call $p(\bar{t})$, there is a tuple $(p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}), \Phi_1)$ in $I_\alpha$ such that $p(\bar{t}) \in \gamma(p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}))$ and $r \in \Phi_1(s)$, where $s = size_p(input_p(\bar{t}))$.

Lemma 1. $p$ is partially correct with respect to $I_\alpha$ if:

- For all $p(\bar{t}) \in C_p$, there is a tuple $(p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}), \Phi_1)$ in $I_\alpha$ such that $p(\bar{t}) \in \gamma(p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}))$, and
- $p$ is partially correct with respect to every tuple in $I_\alpha$.

Definition 6. Given two resource usage interval functions $\Phi_1$ and $\Phi_2$, such that $\Phi_1, \Phi_2 : S \mapsto \mathcal{RI}$, where $S \subseteq \mathcal{R}^k$, we define the inclusion relation $\sqsubseteq_f$ and the intersection operation $\sqcap_f$ as follows:

- $\Phi_1 \sqsubseteq_f \Phi_2$ iff for all $\bar{n} \in S$ ($S \subseteq \mathcal{R}^k$), $\Phi_1(\bar{n}) \subseteq \Phi_2(\bar{n})$.
- $\Phi_1 \sqcap_f \Phi_2 = \Phi_3$ iff for all $\bar{n} \in S$ ($S \subseteq \mathcal{R}^k$), $\Phi_1(\bar{n}) \cap \Phi_2(\bar{n}) = \Phi_3(\bar{n})$.

Consider a tuple $(p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}), \Phi_1)$ in the intended meaning $I_\alpha$, and a tuple $(p(\bar{\tilde{v}}) : c(\bar{\tilde{v}}), \Phi)$ in the computed abstract semantics $\llbracket P \rrbracket_{\alpha+}$ (for simplicity, we assume the same tuple of variables $\bar{\tilde{v}}$ in all abstract objects).

Definition 7. We say that $(p(\bar{\tilde{v}}) : c(\bar{\tilde{v}}), \Phi) \sqsubseteq (p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}), \Phi_1)$ if $c_1(\bar{\tilde{v}}) \subseteq c(\bar{\tilde{v}})$ and $\Phi \sqsubseteq_f \Phi_1$.

Note that the condition $c_1(\bar{\tilde{v}}) \subseteq c(\bar{\tilde{v}})$ can be checked using the CiaoPP capabilities for comparing program state properties such as types or variable sharing.

Definition 8. We say that $(p(\bar{\tilde{v}}) : c(\bar{\tilde{v}}), \Phi) \sqcap (p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}), \Phi_1) = \emptyset$ if $c_1(\bar{\tilde{v}}) \subseteq c(\bar{\tilde{v}})$ and $\Phi \sqcap_f \Phi_1 = \Phi_0$, where $\Phi_0$ is the empty function defined as follows: $\Phi_0(\bar{n}) = \emptyset$ for all $\bar{n} \in S$ ($S \subseteq \mathcal{R}^k$).

Lemma 2. If $(p(\bar{\tilde{v}}) : c(\bar{\tilde{v}}), \Phi) \sqsubseteq (p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}), \Phi_1)$ then $p$ is partially correct with respect to $(p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}), \Phi_1)$.

Proof. If $(p(\bar{\tilde{v}}) : c(\bar{\tilde{v}}), \Phi) \sqsubseteq (p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}), \Phi_1)$ then $c_1(\bar{\tilde{v}}) \subseteq c(\bar{\tilde{v}})$ and $\Phi \sqsubseteq_f \Phi_1$. For all $p(\bar{t}) \in C_p$ such that $r$ is the amount of resource usage of the computation of the call $p(\bar{t})$, it holds that: if $p(\bar{t}) \in \gamma(p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}))$ then $p(\bar{t}) \in \gamma(p(\bar{\tilde{v}}) : c(\bar{\tilde{v}}))$ (because $c_1(\bar{\tilde{v}}) \subseteq c(\bar{\tilde{v}})$), and thus $r \in \Phi_1(s)$, where $s = size_p(input_p(\bar{t}))$ (because of the safety of the analysis). Since $\Phi \sqsubseteq_f \Phi_1$, we have that $r \in \Phi_1(s)$.

Lemma 3. If $(p(\bar{\tilde{v}}) : c(\bar{\tilde{v}}), \Phi) \sqcap (p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}), \Phi_1) = \emptyset$ then $p$ is incorrect w.r.t. $(p(\bar{\tilde{v}}) : c_1(\bar{\tilde{v}}), \Phi_1)$.
3.2 Comparing Resource Usage Interval Functions

During verification/debugging within the framework described in the previous section, we will need to compare abstract tuples following Table 2. Thus, whenever \( c_1(\bar{v}) \subseteq c(\bar{v}) \) we will have to determine whether \( \Phi \subseteq \Phi_1 \) or \( \Phi \cap \Phi_1 = \Phi_0 \).

**Definition 9.** Given two resource usage bound functions \( \Psi_1 \) and \( \Psi_2 \) (as in Definition 1, \( \Psi_1, \Psi_2 : S \rightarrow \mathbb{R}, \) where \( S \subseteq \mathbb{R}^k \)), we define the \( \leq_f \) relation as follows:

\[
\Psi_1 \leq_f \Psi_2 \text{ iff for all } \bar{n} \in S, \text{ it holds that } \Psi_1(\bar{n}) \leq \Psi_2(\bar{n})
\]

where \( \leq \) represents the standard relation between real numbers augmented with the special symbols \( \infty \) and \( -\infty \). Similarly, we define \( <_f, >_f \) and \( \geq_f \).

**Lemma 4.** Given two resource usage interval functions \( \Phi_1 \) and \( \Phi_2 \), we have that:

- \( \Phi_1 \subseteq_f \Phi_2 \) if \( \Phi_1^l \leq_f \Phi_1^r \) and \( \Phi_1^u \leq_f \Phi_2^u \).
- \( \Phi_1 \cap_f \Phi_2 = \Phi_0 \) if \( \Phi_1^l <_f \Phi_2^l \) or \( \Phi_1^u <_f \Phi_2^u \).

**Corollary 1.** Let \( (p(\bar{v}) : c(\bar{v}), \Phi) \) and \( (p(\bar{v}) : c_1(\bar{v}), \Phi_1) \) be tuples expressing an abstract semantics \( \llbracket P \rrbracket_{\alpha_a} \) inferred by analysis and an intended abstract semantics \( I_\alpha \) (given in a specification) respectively, such that \( c_1(\bar{v}) \subseteq c(\bar{v}) \), and for all \( \bar{n} \in S \) \( (S \subseteq \mathbb{R}^k) \), \( \Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})] \) and \( \Phi_1(\bar{n}) = [\Phi_1^l(\bar{n}), \Phi_1^u(\bar{n})] \). We have that:

1. If for all \( \bar{n} \in S \), \( \Phi_1^l(\bar{n}) \leq \Phi^l(\bar{n}) \) and \( \Phi_1^u(\bar{n}) \leq \Phi^u(\bar{n}) \), then \( p \) is partially correct with respect to \( (p(\bar{v}) : c_1(\bar{v}), \Phi_1) \).
2. If for all \( \bar{n} \in S \), \( \Phi^u(\bar{n}) < \Phi_1^l(\bar{n}) \) or \( \Phi^l(\bar{n}) < \Phi_1^u(\bar{n}) \), then \( p \) is incorrect with respect to \( (p(\bar{v}) : c_1(\bar{v}), \Phi_1) \).

When \( \Phi_1^p \) (resp., \( \Phi_1^r \)) is not present in a specification, we assume that \( \forall \bar{n} \) \( (\Phi_1^p(\bar{n}) = \infty) \) (resp., \( \Phi_1^r = -\infty \) or \( \Phi_1^l(\bar{n}) = 0 \), depending on the resource). With this assumption, one of the resource usage bound function comparisons in the sufficient condition 1 (resp., 2) above is always true (resp., false) and the truth value of such conditions depends on the other comparison.

For the particular case where resource usage bound functions depend on one argument, the result of the resource usage bound function comparison in our approach is in general a set of intervals of input data sizes for which a function is less, equal, or greater than another. This is explained in Section 4 and allows us to give intervals of input data sizes for which a program \( p \) is partially correct (or incorrect).

3.3 Dealing with Preconditions Expressing Input Data Size Parameters

Given the formalization presented in the previous sections, note that it is now straightforward to allow checking assertions which include preconditions expressing intervals within which the input data size of a program is supposed to lie
(i.e., intervals for which each assertion is applicable). All that is required is to modify some definitions.

From the practical view, we have extended the Ciao assertion language with the new intervals(A, B) property (to be used in the Precond field of assertions), which expresses that the input data size A is included in some of the intervals in the list B, and we have made the corresponding changes in the algorithms.

To give an example, the element c(\Bar{v}) in a tuple representing an abstract semantics should be allowed to include the property intervals(A, B). Also, the operation c_I(\Bar{v}) \sqsubseteq c(\Bar{v}) in Definition 7 should be extended to deal with data size intervals. However, in practice we can follow a simpler approach. Even if we do have an abstract domain to infer the property intervals(A, B), it is safe to work with a c(\Bar{v}) expressing only instantiation states (since c_I(\Bar{v}) \sqsubseteq c(\Bar{v}) is preserved). Also, in practice, we can perform the assertion checking ignoring the data size intervals in c_I(\Bar{v}), and then ”filter” the intervals of the verification outcome with the intervals initially present in c_I(\Bar{v}) (using interval intersection operations).

4 Resource Usage Bound Function Comparison

As stated in [12, 13], fundamental to our approach to verification is the operation that compares two resource usage bound functions, one of them inferred by the static analysis and the other one given in an assertion present in the program (i.e. given as a specification). Given two of such functions, \Psi_1(n) and \Psi_2(n), n \in \mathbb{R}, the objective of this operation is to determine intervals for n in which \Psi_1(n) > \Psi_2(n), \Psi_1(n) = \Psi_2(n), or \Psi_1(n) < \Psi_2(n).

Our approach consists in defining f(n) = \Psi_1(n) - \Psi_2(n) and finding the roots of the equation f(n) = 0. Assume that the equation has m roots, n_1, \ldots, n_m. These roots are intersection points of \Psi_1(n) and \Psi_2(n). We consider the intervals S_1 = [0, n_1), S_2 = (n_1, n_2), S_m = \ldots \ (n_{m-1}, n_m), S_{m+1} = (n_m, \infty). For each interval S_i, 1 \leq i \leq m, we select a value v_i in the interval. If f(v_i) > 0 (respectively f(v_i) < 0), then \Psi_1(n) > \Psi_2(n) (respectively \Psi_1(n) < \Psi_2(n)) for all n \in S_i.

Since our resource analysis is able to infer different types of functions (e.g., polynomial, exponential, logarithmic and summatory), it is also desirable to be able to compare all of these functions.

For polynomial functions there exist powerful algorithms for obtaining roots, e.g. the one we are using which is implemented in the GNU Scientific Library (GSL) which offers a specific polynomial function library that uses analytical methods for finding roots of polynomials up to order four, and uses numerical methods for higher order polynomials. For the other functions, we safely approximate them using polynomials such that they bound (from above or below as appropriate) such functions. In this case, we should guarantee that the error falls in the safe side when comparing the corresponding resource usage bound functions. We refer the reader to [13] for a full description of how such approximations are performed.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ID</th>
<th>Assertion</th>
<th>Verif. Result</th>
<th>Time (mS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibonacci</td>
<td>A1</td>
<td>:- comp fib(N,R) + cost(ub,steps,exp(2,int(N))-1000).</td>
<td>F in [0, 10]</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ T in [11, ∞]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2</td>
<td>:- comp fib(N,R) + (cost(ub,steps,exp(2,int(N))-1000),</td>
<td>F in [0, 10] ∪ [15, ∞]</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cost(lb,steps,exp(2,int(N))-10000)).</td>
<td>T in [11, 14]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A3</td>
<td>:- comp fib(N,R) + (intervals(int(N),[i(1,12)]))</td>
<td>F in [0, 10]</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ (cost(ub,steps,exp(2,int(N))-1000),</td>
<td>T in [11, 12]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>cost(lb,steps,exp(2,int(N))-10000)).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naive reverse</td>
<td>B1</td>
<td>:- comp nrev(A,B) + (cost(lb,steps,length(A)),</td>
<td>F in [0, 3]</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cost(ub,steps,exp(length(A),2))).</td>
<td>T in [4, ∞]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B2</td>
<td>:- comp nrev(A,,1) + (cost(lb, steps, length(A)),</td>
<td>F in [0, 0] ∪ [17, ∞]</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cost(ub, steps, 10*length(A))).</td>
<td>T in [1, 16]</td>
<td></td>
</tr>
<tr>
<td>Quick sort</td>
<td>C1</td>
<td>:- comp qsort(A,B) + cost(ub, steps, exp(length(A),2))</td>
<td>F in [0, 2]</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ C in (2, ∞)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>:- comp qsort(A,B) + cost(ub, steps, exp(length(A),3))</td>
<td>F in [0, 1]</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ C in (1, ∞)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client</td>
<td>D1</td>
<td>:- pred main(Op, I, B) + cost(ub, bits_received,</td>
<td>C in (0, 8)</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exp(length(I),2)).</td>
<td>T in [0, 8]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D2</td>
<td>:- pred main(Op, I, B) + cost(ub, bits_received,</td>
<td>T in [0, ∞]</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10*length(I)).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D3</td>
<td>:- pred main(Op, I, B) + intervals(length(I),[i(1,10),i(100,plusinf)])</td>
<td>T in [0, 10] ∪ [100, ∞]</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ cost(ub, bits_received,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10*length(I)).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reverse</td>
<td>E1</td>
<td>:- pred reverse(A, B) + (cost(ub, ticks, 10 * length(A) - 20)).</td>
<td>F in [0, 4]</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ T in [5, ∞]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Results of the interval-based static assertion checking integrated into CiaoPP.
5 Implementation and Experimental Results

The resource usage verification techniques presented in this paper have been implemented and integrated in a seamless way within the Ciao/CiaoPP framework that unifies static and run-time verification, as well as unit testing [14].

As mentioned before, for the implementation of the resource usage function comparison operations we have used the GNU Scientific Library [8]. To this end we have defined a Ciao-GSL binding through the native code (C) interface. As mentioned before, we have implemented comparisons for polynomial, exponential, logarithmic and summatory functions, the latter two through safe approximations via polynomials.

We have also performed an experimental assessment of the accuracy and efficiency of the resource usage verification techniques. Table 3 shows some experimental results obtained with our prototype implementation. The column labeled Benchmark shows information about the program to be verified, which includes its name and the upper (ub) and lower (lb) bound resource usage functions inferred by CiaoPP’s static analysers.

The columns ID and Assertion show several assertions expressing resource usages to be statically checked, which are written by the user together with the source code (ID is just an identifier to facilitate discussion.). We can see that some assertions only specify upper bounds (e.g., A1, C1 or C2), and other assertions specify both upper and lower bounds (e.g., A2, A3, B1 or B2). Note also that some assertions include preconditions expressing intervals within which the input data size of the program is supposed to lie (A3 and D3). The column Check Result shows the result of the assertion checking process, which in general express intervals of input data sizes for which the assertion is true (T), false (C) or it has not been posible to determine whether it is true of false (C). Finally, the column labeled Time shows the resource verification times in milliseconds, on a Intel Centrino 1.5 GHz with one processor, 768Mb of RAM memory, running Debian Lenny, kernel 2.6.26-2-686.

Note that we can deal with different types of resource usage functions, as for example polynomial functions (see e.g. programs naive reverse, client, and reverse), exponential functions (see the fibonacci program), and summatory functions (as in the quick sort program).

We can see that in general polynomial functions are faster to check than other functions, because they do not need additional processing for approximation. However the additional time to compute approximations is very reasonable in practice.

Table 4 shows the results of an experiment that we have performed for the case where assertions include preconditions expressing input data size intervals. The experiment consists on comparing the method described so far (referred to as Root in Column Method) with a simple method (referred to as Eval) consisting on evaluating the resource usage functions (i.e., the ones inferred by analysis and the ones present in the assertions) for all the values in a given input data size interval (which is a finite set of natural numbers) and comparing the results. Column ID refers to the assertions in Table 3. Assertion checking
Table 4. Comparison of assertion checking times for two methods dealing with pre-
conditions expressing input data size intervals.

<table>
<thead>
<tr>
<th>ID</th>
<th>Method</th>
<th>[1,12]</th>
<th>[1,100]</th>
<th>[1,1000]</th>
<th>[1,10000]</th>
<th>[1001,10000]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3</td>
<td>Root</td>
<td>84</td>
<td>84</td>
<td>84</td>
<td>84</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>Eval</td>
<td>80</td>
<td>84</td>
<td>132</td>
<td>644</td>
<td>628</td>
</tr>
<tr>
<td>D3</td>
<td>Root</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Eval</td>
<td>36</td>
<td>36</td>
<td>48</td>
<td>116</td>
<td>112</td>
</tr>
</tbody>
</table>

times (in milliseconds) are shown for different input data size intervals (columns under the Intervals label). We can see that checking time grows quite slowly compared to the length of the interval, which grows exponentially.

6 Conclusions

We have presented several extensions and improvements to our framework for verification/debugging (implemented in the CiaoPP system) dealing with specifications about the resource usage of programs, itself and extension of the CiaoPP framework for verification of functional or program state properties. We have provided a full formalization and we have improved the resource usage function comparison method by extending the class of resource usage functions that can be compared and providing better algorithms, which in addition allow for the case when the assertions include preconditions expressing input data size intervals. We have also reported on a prototype implementation and provided the first experimental results, which are encouraging, suggesting that our framework is feasible and accurate in practice.

References


Resource Analysis with Static Values and Probabilities
(Work in Progress Report)

Christoph A. Herrmann and Kevin Hammond

University of St Andrews, Scotland KY16 9SX, UK,
{ch,kh}@cs.st-andrews.ac.uk,
WWW home page: http://www.cs.st-andrews.ac.uk/~ch

Abstract. This paper combines our previous work on cost analysis including symbolic expressions for static program values [8] with probabilities for branching and loop entry. Inclusion of probabilities was motivated by systems with probabilistic behaviour, e.g. interrupts, and algorithms with randomisation, e.g. Markov Chain Monte Carlo methods. We present a concise calculus which, without probabilities, would only be a subset of what counter automata in our previous work can describe. However, this calculus can handle simple block-structured programs. The aim of the imposed restrictions is to exclude that static program values are influenced by probabilistic choices and incur unnecessary complexity of the analysis. It also avoids the need to detect recurrences and makes it easily applicable. We apply our calculus to a small example program in C and compare the outcome of the analysis with a mean of many simulation results.

1 Introduction

We present a new approach which is motivated by the resource analysis of software-controlled systems, in particular for algorithmic processing on embedded systems, using parameters and probabilities. Since algorithms, such as image processing can be computationally expensive and embedded systems, such as autonomous vehicles have small computational capabilities and energy resources, it is important that such an analysis provides estimates for computation time, memory and energy consumption symbolically in parameters. Parameters can refer to the data volume to be processed or control precision of the algorithmic results to meet resource limitations, e.g. that the image processing is fast enough to achieve a given frame rate. Aside from computational resources we could also include resource properties of external devices such as sensors and motors. Without loss of generality, we will focus on the analysis of execution time for code blocks.

Furthermore, for an embedded system input events can occur with certain probabilities, triggering actions that cause quite different costs. Also, some heuristics are using decisions based on the outcome of a random number generator. We therefore provide a means to include probabilities into the analysis.
We intend our analysis to deliver an average case result, without regarding input distributions because this is supposed not to impact the static control structure of the program. However, where we do not have enough information which branch of a program is taken, we take the maximum of the costs. We believe that this will be sufficient to give reliable estimates of an upper bound of the resource consumption for a sufficiently large number of system actions.

The paper is organised as follows. In Section 2 we present a language for abstract specifications of programs. Section 3 explains the cost analysis. In Section 4 we apply our method to a small example. Section 5 discusses related work and Section 6 concludes.

2 A Language for Abstract Cost Specifications

We present a core set of language constructs sufficient to discuss the aspects that occur in a static analysis. The language is constructed in a way that no detection of recurrences or loop variables is required. There is only one data type: rational numbers, which are used to express cost terms and, restricted to the subset of natural numbers: iteration counters or static size information of run-time data.

2.1 Expressions

Expressions form a sublanguage $E$ and can be rational constants, variables, arithmetic expressions and conditionals. $Q$ is the set of rational numbers and $V$ is an enumerable set of variables.

$$E ::= C \mid V \mid E + E \mid E \cdot E \mid E^a \mid \text{IfZ} E E E$$

The value of an expression $e$ is given by $\varepsilon[e] \in V \rightarrow Q$, defined as follows, where $\epsilon \in V \rightarrow Q$:

$$\begin{align*}
\varepsilon[C]i & \epsilon = i \\
\varepsilon[a] & \epsilon = v_a \\
\varepsilon[b] & \epsilon = v_b \\
\varepsilon[a + b] & \epsilon = v_a + v_b \\
\varepsilon[a \cdot b] & \epsilon = v_a \cdot v_b \\
\varepsilon[a :^a b] & \epsilon = v_a \\
\varepsilon[\text{IfZ} a b c] & \epsilon = v_a
\end{align*}$$

Subtraction by a value $s$ can be expressed in terms of addition with the product of the constant (-1) and $s$. Equality comparison between two expressions can be expressed by comparing their difference with 0. A predicate to control Integral counting down to zero also can be expressed this way. Nevertheless, operators such as comparing for being less could useful for other reasons, but are not necessary for the explanations in this paper and would not incur complications other than in the simplification of cost expressions.
2.2 Statements

Using the expression language $E$ the language $S$ of statements contains assignments, cost annotations, sequence, static and dynamic conditional, static loops, probabilistic choice and loop with reentry probability. $P$ is a subset of rational numbers strictly between 0 and 1 used for probabilities.

$$S ::= \text{Ass } V E \mid \text{Cost } E \mid \text{Seq } S S \mid \text{IfZ } E S S \mid \text{RTIf } S S \mid \text{For } V E S \mid \text{PChoice } P S S \mid \text{PLoop } P S$$

We now need to define the denotational semantics of the abstract program information, ignoring run-time values as well as ignoring cost information. This semantics tells us how the values that are included into the analysis are being modified, i.e. it delivers an environment for these values as a result. With $\epsilon[x \rightarrow v]$ we denote a (re-)assignment of the variable $x$ in environment $\epsilon$ with the value $v$, i.e.:

$$\epsilon[x \rightarrow v](y) = \begin{cases} v & \text{if } x = y \\ \epsilon(y) & \text{otherwise} \end{cases}$$

Here, we only present the rules for modifying the environment. The cost information is expressed using separate rules in the next section.

$$\begin{align*}
S[\text{Ass } x e]_\epsilon &= \epsilon[x \rightarrow v] \\
S[\text{Cost } e]_\epsilon &= \epsilon \\
S[\text{Seq } a b]_\epsilon &= S[a]_\epsilon \circ S[b]_\epsilon \\
S[\text{IfZ } c a b]_\epsilon &= S[a]_\epsilon \circ S[b]_\epsilon \\
S[\text{For } v u a]_0 &= S[\text{For } v u a]_\epsilon \\
S[\text{RTIf } a b]_\epsilon &= S[\text{RTIf } a b]_\epsilon \\
S[\text{PChoice } p a b]_\epsilon &= S[\text{PChoice } p a b]_\epsilon \\
S[\text{PLoop } p a]_\epsilon &= S[\text{PLoop } p a]_\epsilon 
\end{align*}$$

The semantics of the $\text{For}$ loop is defined via chaining the modifications of the environment in each loop iteration, where the loop counter $v$ is instantiated with the current iteration. In a symbolic treatment, the loop requires to find a closed form for the modification of the environment within $n$ repetitions (for arbitrary $n$), since this is later part of the cost formula.

All assignments made in the runtime-value dependent branches and probabilistic branches and loops (last three rules) are invalidated at the end of the branch. This is necessary to exclude dependences from run-time values to static values.
For the real program this means that all variables meant to be static information and defined outside the scope of the run-time dependent or probabilistic part must at the end of this part be restored to the state at the beginning, or the analysis will not reflect the program behaviour.

3 Cost Analysis

3.1 Assignments

This covers cases where program variables affecting static conditions and costs are manipulated. Run-time values should not appear in $e$.

$$\overline{C[\text{Ass } x \ e]} \epsilon = 0$$

Assignments themselves do not incur costs, but costs can be attached to them by using the cost annotation below.

3.2 Cost annotations

This occurs in the abstraction in those places where code blocks have already been analysed as a whole (e.g., by AbsInt’s aIT tool) and serves to insert their cost into the analysis.

$$\overline{\epsilon [x]} \epsilon = c$$

$$\overline{C[\text{Cost } x]} \epsilon = c$$

3.3 Sequence

The sequence is the first case we encounter where the cost formula depends on the modification of the environment by the first component. Still our method is purely compositional as the structurally recursive derivation rules demonstrate.

$$\overline{C[a]} \epsilon_0 = c_a$$
$$\overline{S[a]} \epsilon_0 = \epsilon_1$$
$$\overline{C[b]} \epsilon_1 = c_b$$

$$\overline{C[\text{Seq } a \ b]} \epsilon_0 = c_a + c_b$$

3.4 Static conditional

$$\overline{\epsilon [p]} \epsilon = 0$$
$$\overline{C[a]} \epsilon = c_a$$

$$\overline{\epsilon [p]} \epsilon \neq 0$$
$$\overline{C[b]} \epsilon = c_b$$

$$\overline{S[\text{IfZ } p \ a \ b]} \epsilon = c_a$$

$$\overline{S[\text{IfZ } p \ a \ b]} \epsilon = c_b$$
3.5 Conditional branching on non-static information

If the branch taken depends on run-time values, the best the analysis can do is to deliver the maximum cost of both branches to achieve a safe approximation of resources within upper bounds, because it might well be that always the branch with the higher costs is taken. This is a situation in which we need to depart from an accurate average case analysis, but we believe that this treatment is appropriate for serious resource management. Note that if the branch depends only on a fixed probability, the probabilistic choice can be used instead to obtain more precise results. With ↑ we denote the maximum of two rational values.

\[
\frac{C[a]}{\epsilon} = c_a \quad \frac{C[b]}{\epsilon} = c_b
\]

\[
C[RTIf a b]\epsilon = c_a \uparrow c_b
\]

3.6 Static Loop

The For loop requires that the rational number which specifies the number of iterations belongs to the subset of natural numbers. The analysis has to check this, in simple cases by subtyping, in more complicated cases by checking properties such as (\(n \in \mathbb{N} \Rightarrow \frac{n \cdot (n+1)}{2} \in \mathbb{N}\)).

\[
\begin{align*}
\forall_{i=0}^{n-1} (S[a](\epsilon_i[v \mapsto i]) = \epsilon_{i+1} \land \frac{C[a]}{\epsilon}(\epsilon_i[v \mapsto i]) = c_i) \\
C[For \ v \ u \ a]\epsilon_0 = \sum_{i=0}^{n-1} c_i
\end{align*}
\]

As we have discussed before, the modification of the environment requires us to obtain a closed for a repetitive execution of the body. Another complication is the summation of costs. If all \(c_i\) are the same, we can just deliver a product of \(n\) and \(c_0\). Otherwise we need to perform a symbolic summation [4].

If we need the result only for a particular value of \(n\) and this value is small enough, then there is also the possibility of unfolding the application of the \(\forall\) and \(\sum\) quantifiers.

3.7 Probabilistic Choice

When the original program branches with a particular probability, the following rule models the expected probability.

\[
0 < p < 1 \quad \frac{C[a]}{\epsilon} = c_a \quad \frac{C[b]}{\epsilon} = c_b
\]

\[
C[PChoice p a b]\epsilon = p \cdot c_a + (1-p) \cdot c_b
\]

In a nested application, the dependencies will result in a multiplication of the probabilities.
3.8 Probabilistic Loop

Our language includes a special kind of loop which is (re-)entered with a probability of $p \in (0, 1)$. Two situations in which this can occur is (1) a cyclic process which is interrupted or fails with a certain probability or (2) a while loop which performs random testing, where the probability of success is known.

For the analysis of such a loop we consider that the probability for having exactly $n$ iterations is $p^n \cdot (1 - p)$ which can be easily proved by induction on $n$: we have a sequence of entering the loop $n$ times each of probability $p$ followed by leaving the loop with a probability of $(1 - p)$.

We can then calculate the mean number of body executions by:

$$\sum_{n=0}^{\infty} n \cdot P(#\text{iterations} = n) = \sum_{n=0}^{\infty} n \cdot p^n \cdot (1 - p) = (1 - p) \sum_{n=0}^{\infty} n \cdot p^n = (1 - p) \cdot \frac{p}{(1 - p)^2} = \frac{p}{1 - p}$$

We obtain the rule for the probabilistic loop:

$$c[\text{PLoop } p \ a]e = c \cdot \frac{p}{1 - p}$$

As long as we use only constant values for the probability, the fraction $\frac{p}{1 - p}$ will appear as a rational coefficient of a polynomial. But if we permit a variable and the denominator cannot be factored out easily (e.g., because $p$ depends on a loop counter) it would be necessary to deal with piecewise rational functions in the symbolic simplification.

4 Example

Our example is an abstraction of a system which performs a number of iterations. At the beginning of each iteration it checks whether an interrupt has been occurred which is then handled. Afterwards the code for the body of the iteration, which cannot be interrupted, is executed. In the following code, interrupts are supposed to happen with a probability of 7%. If an interrupt occurs, action has to be taken for a number of $i$ elements. We assume that the costs for functions handle(.) and loop_body can be analysed without much interaction with a program analyser, such as aiT[1]. These costs are then inserted into the cost annotations in the specification.
for(i=0;i<n;i++)
{
    if (interrupt()) // probability = 7%
        for(j=0;j<i;j++)
            handle(j); // cost = 5
    loop_body(); // cost = 50
}

We specify this in the language $S$ as follows:

For "$i$" (V "n")
(Seq (PChoice (7/100) (For "$j$" (V "i") (Cost (C 5)))
        (Cost 0))
      (Cost (C 50))

Our calculus delivers as a cost

$$\sum_{i=0}^{n-1} \left( \frac{7}{100} \cdot \left( \sum_{j=0}^{i-1} 5 \right) + (1 - \frac{7}{100}) \cdot 0 \right) + 50$$

which can then be simplified to:

$$\frac{7}{40} \cdot n^2 + (49 + \frac{33}{40}) \cdot n$$

Compared to a mean of 100000 example profiling executions we obtain:

<table>
<thead>
<tr>
<th>n</th>
<th>analysed</th>
<th>measured</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>515.75</td>
<td>515</td>
<td>0.50%</td>
</tr>
<tr>
<td>100</td>
<td>6732.5</td>
<td>6731</td>
<td>0.02%</td>
</tr>
<tr>
<td>1000</td>
<td>224825.0</td>
<td>224644</td>
<td>0.08%</td>
</tr>
</tbody>
</table>

5 Related Work

Our work originally started with worst-case execution time analysis. Although many existing approaches [13] considered context information in the analysis, this information is often, e.g. by the aiT tool, expressed only by a finite case distinction with a limit to be specified by the user. We believe that a full symbolic analysis is necessary to yield arbitrarily scalable estimates in our application context [8] and the solver or simplifier of the cost expression must permit several variables, case distinctions and arithmetic expressions beyond linearity.

This paper combines two different streams of parametric value analysis and parametric probabilistic analysis. We are not aware of other work dealing with such a combination.
5.1 Parametric Value Analysis

In order to achieve a realistic analysis of a program one needs to regard context information (program modes) [9] and also numerical range information [3].

Two major kinds of languages to describe parametric cost expressions are parametric integer programming [11] and piecewise polynomials [12].

In our own previous work we have analysed programs in Hume [6] via abstract interpretation and translation to counter automata [8], obtaining piecewise polynomial cost expressions in several parameters.

5.2 Parametric Probabilistic Analysis

Inclusion of probabilistic branches was motivated by the need to analyse a program for LiDAR image processing based on a reversible-jump Markov Chain Monte Carlo method [7]. A paper on probabilistic analysis of parametric Markov models inspired us to include a solution for a loop controlled by a reentry probability [5].

Stochastic Model Checking [10] can regard costs associated with state transitions, but this approach does not regard impact of algorithmic components on state transitions as we are undertaking.

Our probabilistic loop might incur the need to deal with functions beyond piecewise polynomials. Treatment of such functions, in particular comparison, has been investigated in the COSTA approach [2].

6 Conclusions

We have demonstrated how to combine probabilistic branches and loops with a static analysis that includes values of iteration counters.

Our structurally recursive rules demonstrate compositionality of our analysis. It is just that there are several aspects required in the analysis, namely expressions, environments, and costs.

Our specification language currently contains only a few constructs and probabilities can only be constants, which permits us to perform symbolic computations in the language of piecewise defined polynomials. To be able to deal with variables appearing in the probability of a loop entry, it would be necessary to extend the expression language to rational functions, i.e. fractions of polynomials.

Simple, non-recursive programs in block-structured languages such as C can be modelled directly, because our specification language supports conditionals and loops with static bounds. We believe that our approach is therefore suited for the resource-aware development of simple embedded systems with kernel-based algorithmic components such as image recognition or signal processing [7].

More advanced programming languages such as Hume [6], providing higher-order functions and asynchronous communications require a more flexible approach. In another paper we have presented an approach based on counter automata [8] which supports this, but lacks support for probabilistic treatment.
Future research could deal with an extension of the method to less structured systems, e.g. counter automata with probabilities, and the automatic extraction of specifications from real programs, e.g. programs written in mHume, a restriction of Hume without complex data structures and higher-order functions. The simplicity of the approach presented here would also encourage an extension for large systems of cooperating agents, as they are occurring in scenarios with multiple autonomous vehicles.

Acknowledgements

This work has been supported by EPSCR grant EP/F030657/1 (Islay) and EU Framework VII project 248828 (ADVANCE).

References


Formal Executable Semantics for Timing Analysis *

Mihail Asavoae Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iaşi, Romania
\{mihail.asavoae,dlucanu\}@info.uaic.ro

Abstract. This paper proposes a general methodology for WCET analysis using the formal executable semantics of the underlying programming language. In our case, the targeted programming language is an integer subset of the Simplescalar PISA instruction set. We work with the K framework, a specialization of rewriting logic for design and analysis of programming languages and obtain the executable semantics specification. Our method relies on the concrete semantics and uses it to derive useful abstractions. We also plug in descriptions of instruction caches and the main memory and perform WCET estimation via reachability analysis. This work proposes, what we think to be, the first attempt to use the formal executable semantics of an assembly language to compute timing information of programs, in particular the WCET.

1 Introduction

Knowledge of program execution time bounds is important in the context of design and verification of embedded real-time systems. Such systems interact with the external environment, yielding a set of real-time constraints that ensures the correctness of the design. In the verification process, it is important to know a priori tight upper bounds on worst-case execution time (WCET) [22] of hard real-time software components of the system.

One may distinguish between two distinct approaches for the worst-case estimation problem: the high-level and the low-level analysis. Used mostly in the early stages of the embedded systems design, a high-level analysis provides the results without considering the architecture description of the system. Its main application is in the hardware/software co-design, where the designer may use WCET information to decide which components should be implemented in hardware. Low-level analysis considers both the program, usually at the assembly language level, and a model of the underlying architecture. The expected results should be accurate to ensure that the timing requirements are satisfied. In the context of low-level analysis, the problem at hand is the WCET estimation of a given program on a given processor. Thus, two important issues should be

* This work has been supported by Project POSDRU/88/1.5/S/47646 and by Contract ANCS POS-CCE, O2.1.2, ID nr 602/12516, ctr.nr 161/15.06.2010 (DAK).
addressed: the longest path search and the micro-architecture modeling. The longest path analysis returns the sequence of instructions that will be executed in the extreme scenario. The micro-architecture modeling describes the hardware system that the program is executed on and determines the WCET of a known sequence of instructions. Most successful approaches include, among others, the use of integer linear programming, ILP, for the longest path [21, 11] and the use of abstract interpretation for micro-architecture modeling [8, 10], or the use of ILP for both the program and micro-architecture consideration problem [12].

We propose a novel approach, based on the K framework [20, 5], to help finding a safe and tight low-level WCET estimation. Our approach works on an assembly programs, obtained from disassembling the Simplescalar [3] executable files. The choice of Simplescalar toolset for our proposed WCET analysis approach is inspired from [11]. We use the K framework to define the executable semantics of the language of interest and we plug in the K description of various micro-architecture such as caches, pipelines or hardware speculation techniques. In this way, we obtain an executable semantics of a program that runs on a specified processor. We mention here a few advantages that our proposed method has over the existing approaches. The first advantage is the possibility of executing the program semantics to derive the information of interest, while the second is the possibility of having distinct semantics for the program and the underlying architecture within the same framework. Moreover, the method facilitates timing bounds computation in a compositional framework. To the best of our knowledge, this is the first rewrite-based approach in the context of worst-case timing estimation.

We define the integer subset of Simplescalar [3] PISA assembly language, that we call SSRISC assembly language. We present a number of SSRISC instructions, grouped in arithmetic-logic instructions, branch and jump instructions, load and store instructions and a special instruction for program errors - break. The arithmetic-logic group includes signed addition with overflow, with register-based and immediate operands, instructions for logical and/or, multiplication and division, the latter posing the "divide-by-0" issue. The second group of instructions include the infamous indirect branch, with the instruction jr Rs, as well as branch instructions and the unconditional jump. The load and store instructions present displaced and indexed addressing variants, and corroborated with the other instructions in the language, we have all the possible addressing modes covered.

Related Work. We elaborate next on using model checking for WCET estimation, as it is the backbone of our proposed approach. The first use of model checking technology in WCET estimation is introduced in Metzner’s paper [16], where a processor with simple cache is checked via multiple runs, selected in a binary search fashion. More recent works are around the UPPAAL model checker [2], where the control flow graph of the program and various micro-architecture features are represented as timed automata. In [13], the focus is on multicore systems with timing information extracted from the TDMA and FCFS memory bus models, whereas [6] proposes modular representations of micro-
architecture of several ARM processors. In both papers, UPPAAL explores the timed automata models and the WCET of a program is extracted from the clock constraints of these automata.

The Maude system [4] is the implementation of rewriting logic and, together with a number of integrated methodologies and tools such as a reachability states exploration tool, an LTL model checker, an inductive theorem prover, ITP, as well as other specialized checkers, it enables specification and analysis of programming languages. In the context of rewriting logic, [9] proposes a first approach to model micro-architecture elements and low level languages. The Real Time Maude [17] specializes Maude for design and analysis of real-time and hybrid systems, with support for simulation using timed rewriting and for verification using a time-bounded version of the LTL Maude model checker. Real Time Maude has been used to specify and verify a wide range of applications, such as object-oriented real-time languages, cryptographic protocols or discrete-event models.

The K framework, described in [20], supports the definitions of programming languages using a specialized notation for manipulating program configurations. K shows its versatility when handling definitions of real languages, such as C in [7], Scheme in [14], Verilog in [15] as well as definitions for type systems, a Hoare style program verifier [19] or a model checker with predicate abstraction [1]. K-Maude tool [5] implements the K framework on top of Maude system and provides, in this way, access to all Maude’s aforementioned supporting tools.

To the best of our knowledge, the methodology we propose is novel in several regards. First, with respect to existing WCET estimation approaches, this is the first use of the formal executable semantics of the underlying language to derive convenient abstractions. We exemplify how, starting with the definition (of a subset) of an assembly language, we obtain, within the same framework, a very simple abstract semantics. We also plug in definitions for instruction caches and rely on Maude’s reachability states exploration capabilities to compute WCET bounds for programs in the presence of micro-architecture features that we mention. Second, with respect to rewriting logic in general and K in particular, this is the first use of rewriting-based techniques for WCET analysis. While [9] presents a first encoding of an instruction set architecture in rewriting logic, our approach, taking advantage of the underlying modularity of K definitions and proposes the first parametric description of instruction caches behaviors in rewriting-based environment. Our description is similar to the specification achieved using timed automata in [6], without the pipeline behavior that we have not implemented yet. Third, with respect to the K framework alone, our work is the first definition of an assembly language and micro-architecture features, as well as, the first definitional development of a WCET analyzer.

Outline of the paper. The paper is organized as follows. Section 2 uses a subset of the Simplescalar assembly language to introduce K, a specialized framework for design and analysis of programming languages. Section 3 presents the K definitions for the concrete formal executable semantics and how to derive from this a symbolic semantics. Section 4 proposes an integrated system...
for WCET estimation, based on the formal executable semantics and refined with micro-architecture considerations. Section 5 introduces a K definition for a WCET analysis, that uses the underlying Maude system technology. The last section contains the conclusions.

2 The K Framework

The K framework emerged from rewriting logic to help in design and analysis of programming languages. A K specification consists of configurations, computations and rules. The configurations, formed of K cells, are (potentially) labeled and nested structures that represent program states. The rules in K are divided into two classes: computational rules, that may be interpreted as transitions in a program execution, and structural rules, that modify a term to enable the application of a computational rule. The K framework allows one to define modular and executable programming language semantics.

We give practical insights into the K framework when we use it to define the integer subset of Simplescalar [3] PISA assembly language, which in turn has been inspired from MIPS IV. We call this language SSRISC. Whereas, the entire subset of integer-based instructions has been implemented, for the sake of the presentation, we describe only a snippet with arithmetic-logic instructions, branch and jump instructions, load and store instructions as well as a special break instruction for program errors. Apart from the instruction set presented in [3], a number of pseudo-instructions appears in the executables that we analyze, therefore they are encoded as well. For presentation purposes, we omit to describe the pseudo-instructions, as well.

The general methodology for language definitions in K begins with the (abstract) syntax, establishes the structural elements that make the configuration and then gives the semantic rules. We discuss the first two in this section and present the K rules for SSRISC semantics in the next section.

The K syntax with annotations for a subset of SSRISC assembly language is given, using actual K code, in Fig.1. The left column shows the abstract syntax, in BNF form, while the right column introduces a special K notation, called strictness attribute. strict is a reserved keyword in K and informally, it means that the particular operand on which strictness apply is reduced, using the register look-up in Fig.3, to a base value, called KResult. More formally, this reduction is via a set of generated heating/cooling equations empowered with context reduction style evaluation. For example, the add instruction is strict on the second and third operands, which implies that the last two registers, called sources, are reduced to values before the actual addition takes place and the first, destination register gets this value. The addi instruction has an immediate operand, Imm, therefore it is strict only in the second argument. When strict appears without arguments, i.e. for mult instruction, it means that all the operands reduce to KResult values, before further processing.

The program configuration is a wrapped multiset of cells, written as ⟨cont⟩lbl where cont is the cell content, also a multiset of cells, and lbl is the cell label.
Instr ::= add Reg, Reg, Reg; [strict (2 3)]
addi Reg, Reg, Imm; [strict (2)]
mult Reg, Reg; [strict]
div Reg, Reg; [strict]
and Reg, Reg, Reg; [strict (2 3)]
or Reg, Reg, Reg; [strict (2 3)]
j Addr; [strict]
jr Reg; [strict]
beq Reg, Reg, Addr; [strict (1 2)]
bne Reg, Reg, Addr; [strict (1 2)]
lw Reg, Off (Reg); [strict (3)]
lw Reg, (Reg + Reg); [strict (2 3)]
sw Reg, Off (Reg); [strict (3)]
sw Reg, (Reg + Reg); [strict (2,3)]
break;

Fig. 1. SSRISC language syntax in BNF form (left) with their strictness attributes (right). Reg, Addr, Imm and Off are of sort Int32

These K cells represents the necessary entities, i.e. registers, code memory, data memory etc, to capture the programming language semantics. The K framework uses two special labels: top ⊤ - for the cell that encloses all the other cells, and k - for the computation cell. The SSRISC language configuration is:

Configuration ≡ (K)k ⟨Reg⟩pc ⟨Reg⟩lo ⟨Reg⟩hi
⟨Map[Reg ↦ Val]⟩regs ⟨Val⟩break

The k cell has a special meaning in K, maintaining computational contents, much as programs or fragments of programs. The computations, i.e. terms of special sort k, are nested list structures of computational tasks. Elements of such a list are separated by an associative operator ”↷”, as in s₁↷s₂, and are processed sequentially: s₂ is computed after s₁. We use “.” for the identity of ”↷”. pc is a special register, called program counter and its value indicates the current executing instruction. We opt to represent the program counter in a different cell than the other registers, as it improves the readability of the semantics, especially on conditional and unconditional jumps. lo and hi are special registers, required by the mult and div instructions to hold parts of the computed results. The regs cell contains all the other registers and is a mapping from register names to stored values. The program requires, as well, a representation of the main memory, that holds both the program and the necessary data. We detach the memory modeling from that of the registers as we plan to keep our specification modular. We introduce the instruction caches and further refine the main memory into the code and data memories. We elaborate on this in section 5. The break cell is used, in the strict sense, by the instruction break and, in the more general sense, to capture program errors such as overflow or division by zero.
3 The Formal Executable Semantics

This section introduces the concrete formal executable semantics for the SSRISC assembly language and shows how to derive useful abstractions out of it. As an example, we define a very simple data abstraction using unknown values for program variables.

3.1 The Concrete Semantics

We introduce the K rules by means of defining the semantic rules of SSRISC. The K rule is a generalization of the usual rewriting rule, in the sense that the K rule manipulates only parts of the rewrite term, in three different ways: read, write and don’t care. K proposes a bidimensional representation of a rule, with the lefthand side placed above a horizontal line and the righthand side placed below.

We capture the execution of each SSRISC instruction in a number of successive steps, that start with the instruction, requested from the instruction cache or memory, and, if necessary, get the operands from the data cache or memory, then do the actual processing and end with updating the machine state. Our design target is to capture the language semantics in a correct and concise manner, and, as a result, we propose a single rewrite rule per each SSRISC instruction. To achieve this, we extract some common functionality, as general register lookup and update, or use some wrappers as for the pc register update. Before we describe the instruction semantics, we cover these general rules.

For example, to execute the instruction at the current program point, denoted by the value of PC in the pc cell, the rule in Fig.2, the computation evolves to geti(PC) which is a request from instruction cache or memory. This K rule reads like this: the underlined k cell emphasizes a "write" part of the term and information below, geti(PC) replaces the one above, \( \cdot \). This means that a previously empty computational task list gets the geti task. The pc cell represents a "read" part of the term, it is not underlined, and provides the value PC that is used by geti.

The lookup and update operations on the registers require two cells, k and regs. If the current computational task is a register lookup, for a register R, as show in the first rule in Fig.3, the resulted configuration has the corresponding value I for R from the register file. This K rule brings a new element of the K notation, the "don’t care" part of a term, represented with dots. If these follow a subterm, such as register R in k cell, it means that R is the first computational task, followed by potentially other computational tasks, which are not important in this rule. Similarly, if the top computational task is to update a register, say Rd with a computed value I, the previous value of Rd, denoted by the wildcard \( \_ \), is replaced by I, as shown in the second rule in Fig.3. The dots in the regs cell represents that the element \( R \mapsto I \) is not necessarily the first nor the last in the cell.

The pc register update consists of the following three cases as shown in Fig.4. The first rule represents the automated incrementation before an instruction is
fetched. The second rule addresses the case of a mandatory jump and updates the pc with a specified target address, NewV. The last rule represents the fall-through case of a branch instruction and leaves the value of the pc register only with the previous normal incrementation.

All the K rules for the arithmetic-logic (ALU) instructions, in Fig.5 transform the task in the k cell into a register update, using either updateReg or updateLo/Hi. The former takes two arguments, the register and the value to be written, whereas updateLo and updateHi require only the value. The add and div instructions have extra checks as they could lead to errors, from overflow and respectively division by zero. Therefore, the first K rule states that the add instruction with the source registers having values \( V_1 \) and \( V_2 \) reduces to an overflow check for the signed addition between these two values and, if necessary, followed by the destination register Rd update with the result. The div instruction in the k cell reduces to a division by zero check for the denominator value and, if necessary, followed by the updates for the lo and hi registers.

The branch and jump instructions, in Fig.6, transform the task in the k cell into a correct pc register update, which has the next instruction program counter, as a result of instruction fetch. All these K rules use the setPC operation with the first argument 1 overwrites the value in the pc and 0 to leave it unchanged. The first rule, for the direct jump and the second rule, for the indirect jump change the program counter register with the address Addr, respectively with the value in the register Rs. For the branch when equal beq and branch when not equal bne instructions, the address of the next instruction depends on the comparison between two values, \( V_1 \) and \( V_2 \), fall-through for 0 or branch taken for 1.

The K rules for load and store are in Fig.7. Both load instructions are reduced to memory read requests via getd operation, which takes two arguments, the memory address and the destination register Rd. Similarly, the store instructions are reduced to memory write requests via putd operation, which has the memory address and the source register Rd as arguments. The memory address is computed either using the displaced or the indexed addressing, depending on how the address is calculated. The caches and the main memory, presented in section 4 process the getd and putd requests.

The last semantic rule of the SSRISC language treats the special break instruction. The k cell gets the last term that ends the computation, while the special break cell updates to reflect a program error. We mention that last is also used for normal termination of computation.
3.2 The Abstract Semantics

Usually, the low-level WCET estimation is transformed into the path analysis problem and the micro-architecture modeling. The former works with the control flow graph of the program and, via clever abstractions, attempts to eliminate infeasible paths. One of the strengths of our approach is the possibility of using the concrete formal executable semantics to derive such abstract semantics. To exemplify we describe one simple data abstraction.

We consider the case where all the input program variables are initialized with an unknown value, that we name \texttt{syint32}. The abstract semantics could and should be refined, but it is presented here as the first attempt to have, within the same framework, both the concrete and abstract semantics of the language.

The abstract semantics is derived out of the concrete semantics. Since the program manipulates an abstract value, the domain of the program variables is extended with this special value, \texttt{syint32}. This action keeps all the K semantics rules unchanged, but triggers further extensions in the support operations. For example, the addition between two concrete values, denoted by $+_{\text{Int32}}$, is extended to handle \texttt{syint32} abstract value such that, if any of the operands is symbolic, the result is symbolic. This abstraction enables all the possible executions, as the K rewrite rules for the branch instructions, in Fig.6 rely on a comparison between, potentially symbolic values. For this particularly simple abstraction, the only modifications that are required apply to the underlying built-in operations.

4 Micro-architecture Modeling

Modern processors feature aggressive optimizations that influence the execution of programs. WCET estimation in the presence of micro-architecture be-
Formal Executable Semantics for Timing Analysis

Rule: \( \langle \text{add } R_d, V_1, V_2; \rangle_k \)
updateReg\( (V_1 + \text{Int32} V_2, R_d) \)

Rule: \( \langle \text{addi } R_d, V_1, V_2; \rangle_k \)
updateReg\( (V_1 + \text{Int32} V_2, R_d) \)

Rule: \( \langle \text{mult } V_1, V_2; \rangle_k \)
updateLo\( (V_1 \times \text{Int32} V_2 \mod \text{Int32} 1 \ll \text{Int32} 32) \sim \text{updateHi}(V_1 \times \text{Int32} V_2 / \text{Int32} 1 \ll \text{Int32} 32) \)

Rule: \( \langle \text{div } V_1, V_2; \rangle_k \)
updateLo\( (V_1 / \text{Int32} V_2) \sim \text{updateHi}(V_1 / \text{Int32} V_2) \)

Rule: \( \langle \text{and } R_d, V_1, V_2; \rangle_k \)
updateReg\( (V_1 \& \text{Int32} V_2, R_d) \)

Rule: \( \langle \text{or } R_d, V_1, V_2; \rangle_k \)
updateReg\( (V_1 | \text{Int32} V_2, R_d) \)

Fig. 5. Semantics rules for SSRISC ALU instructions

Rule: \( \langle j \text{ Addr}; \rangle_k \)
setPC\( (1, \text{Addr}) \)

Rule: \( \langle jr \text{ Rs}; \rangle_k \)
setPC\( (1, \text{Rs}) \)

Rule: \( \langle \text{beq } V_1, V_2, \text{Addr}; \rangle_k \)
setPC\( (\text{Bool2Int} (V_1 = \text{Bool} V_2), \text{Addr}) \)

Rule: \( \langle \text{bne } V_1, V_2, \text{Addr}; \rangle_k \)
setPC\( (\text{Bool2Int} (V_1 \neq \text{Bool} V_2), \text{Addr}) \)

Fig. 6. Semantics rules for SSRISC branch and jump instructions

comes harder, as micro-architecture introduces difficult to predict or even non-deterministic behaviors. In the context of WCET estimation, the modeling of instruction and data caches and in-order pipelines have been the most popular [8, 10–12]. We plan to cover a modular design for instruction caches and a simple main memory model. Since modularity is our modeling target, we expect to be able to plug-in various micro-architecture elements, without changing the programming language definition.

Our design relies on a number of modules, corresponding to the "processor" (semantics), instruction cache and main memory, and which communicate using predefined message names. Some of these were informally introduced, like \text{geti} in Fig.2 or \text{getd} and \text{putd} in Fig.7. Next, we overview our module system and elaborate on the instruction cache and main memory implementations in K.

Our system for WCET analysis is built around the definition of the SSRISC assembly language. We recall that the concrete configuration, described in Section 3, omits a store or memory cell, necessary to capture program executions. A first design decision was to let the K rules for the language semantics update
Fig. 7. Semantics rules for SSRISC load, store and break instructions

Fig. 8. K-rules for an instruction request from the main memory

only the registers. In this way, the representation of the memory system is disconnected from register updates and therefore, amenable to refinements. Since we work on an assembly language definition, we emulate the organization of an assembly file into a data and a code text. The K framework for the main memory configuration is presented next.

\[
\text{MemConfig} \equiv (K)_{k} \langle \text{Map[Addr} \mapsto \text{Instr}]_{cmem} \rangle_{cmem} \langle \text{Map[Addr} \mapsto \text{Data}]_{dmem} \rangle_{dmem}
\]

The k cell processes the requests for both instruction or data that come from the cache memories or the processor. The latter is abstractly represented by our language definition. In our initial design, the "processor" issues the instruction requests using the \text{geti(}PC\text{)} operation. The memory system interprets the \text{PC} value as an address and checks the code memory part, the \text{cmem} cell, for it. There are two possible cases, modeled with K rules in Figure 8. First, the instruction is found in the code memory \text{cmem}, and \text{geti(}PC\text{)} rewrites to the actual instruction, while the control is back to the processor. Second, if the instruction is not found, a special token, \text{last}, signals the execution termination. We rely on a special built-in function \text{\$hasMapping} to check if the instruction exists in the code memory.

In a similar fashion, the main memory receives requests for data read with the \text{getd(}Addr\text{)} message, or data write with the \text{putd(}Addr, Data\text{)} message. The data memory cell is checked using \text{Addr} value and, if necessary, updated with \text{Data} value.
We refine the initial design to accommodate an instruction cache specification. The main memory gets now the message from the instruction cache only in the case of a cache miss. The two rules in Fig.8 are the same, except the request message is changed to \texttt{imiss}(PC) to reflect the cache presence. The instruction cache configuration also has a \texttt{k} cell to get the instruction requests from the "processors" - our language definition - and forward requests to the main memory module, in case of a cache miss.

\begin{align*}
\text{ICConfig} & \equiv \left( K \right)_k \left( \Map{Addr \mapsto Instr} \right)_{ic} \left( \Map{ICParam \mapsto Val} \right)_{param} \\
& \left( \Map{PC \mapsto Instr} \right)_{replace} \left( \Map{Hit \mapsto Miss \mapsto Value} \right)_{profile}
\end{align*}

The \texttt{ic} cell is the instruction cache and contains information of the form $Addr \mapsto iwrap(PC, Instr)$, where $Addr$ is the cache address that holds the value of the instruction $Instr$ at the program point $PC$ in the program. A number of parameters such as cache size, cache line size and associativity characterize the cache memories. The cache size is the total number of bytes that can be stored. The cache line size is the number of bytes that can be transferred to and from the memory in one step. The associativity describes the relation between cache lines and memory blocks. A memory block can reside anywhere in the cache, in a group of cache lines or in exactly one line. This leads to the standard terminology of fully associative caches, for the first case, N-associative caches for the second - with N the number of cache lines and the direct-mapped caches for the last case. We assume the particular case of a cache line and a memory block having the same size. The \texttt{replace} cell captures, in a modular way, a number of possible replacement policies. The basic idea is to use a copy of the cache memory that is annotated with various "age" information for the classical FIFO and LRU policies. We recall that a replacement policy decides which memory block is evicted from cache. The last cell, whereas it is not necessary for our timing analysis, keeps profiling information about the number of cache hits and cache misses. For this purpose, the \texttt{profile} cell contains only two counters.

The instruction cache modeling has access to the global system configuration that includes the time cell, $t$. In the strict context of instruction cache influence over timing bounds, this time cell is updated with a hit time or a miss penalty, according to the actual instruction requests. We work under the assumption that a cache hit is 1 time unit and a cache miss is 10 time units. The K rewrite rule for the case of the instruction cache hit is in Fig.9.

The \texttt{k} cell has to process the instruction request, $\texttt{geti}(PC)$, and, if it hits in the cache, the new computational tasks include the program counter normal incrementation, with $\texttt{inc}(PC)$, followed by the instruction, $Ins$. The \texttt{ic} cell contains, at address $X$, this particular instruction. Since our cache modeling is parameterized by cache capacity, line size and associativity, all of these are in cell \texttt{param}, the hit/miss decision is based on these parameters. This makes the K rule conditional, with \texttt{checkCache} operation returning the set of all possible cache lines where the instruction $PC$ could reside. The timing information, in the
cell $t$ is updated with the hit time value. Also, the profiling information updates the number of instruction cache hits, the $hiti$ value with one.

The case of an instruction cache miss poses additional problems, as the replacement policy has to be considered. The configuration has a special cell called replace that actually maintains a shadow copy of the cache with "age" information to enable a parametric implementation of two of the most popular replacement policies: FIFO (round-robin) and LRU. The time value sets the instruction "age" attribute. One particular difference is that LRU policy changes the "age" attribute on both hits and miss, whereas FIFO policy on miss only. For more discussion on cache replacement policies in the context of timing analysis, we refer [18].

Currently, our implementation handles only FIFO replacement policy. The cache miss scenario consists of the following steps: the "processor" request is labeled as miss and the main memory gets the $imiss$ message. After the instruction is retrieved from the main memory, we update the time value with the miss penalty and increment the number of cache misses. The interesting part is the implementation of the cache replacement policy. When the "processor" request is forwarded to the main memory, the replace cell is populated with cache lines. The following situations are possible: the fully associative cache implies that the entire cache content is copied in replace, the direct-mapped has only one cache line copied, the N-associative cache has a group of N cache lines copied. The eviction takes place according to the cache information "age".

5 Reachability Analysis Using Maude System

We work with the current implementation of the K framework, called K-Maude. It is developed on top of Maude system and, in this way, it has access to all verification tools that Maude offers: a command for reachability analysis, an LTL model checker, an inductive theorem prover. We choose the first method, and perform reachability analysis on our K specifications, to determine the WCET bounds. K-Maude takes K specifications and generates rewrite theories in Maude. A rewrite theory has an underlying equational theory, containing equations and membership statements, plus rewrite rules. A rewrite theory defines an abstract transitional system, with the equations giving, via equivalence classes, the states while the rewrite rules give the transitions between these classes. If the lefthand side of a rewrite rule matches the (fragment) of a current state, and the rule
condition is satisfied, the system evolves into the state of the righthand side of the particular rewrite rule. Maude system offers the possibility of unfolding this transition system and proving properties, or getting counterexample information.

To obtain a safe WCET bound for a given program, we need to consider all the possible program executions, implicitly or explicitly. The transition system that a rewrite theory provides could be unfolded using the special search command. The class of hard real-time programs assume that program terminates and, in our formal semantics, we denote the final computational task with the token last. Therefore, all the program executions should terminate in a state that has last in the k cell. A single program execution can terminate in several ways, we present next two such situations. First, there is the normal termination, when there are no more instructions to be executed. This case is described in the second rule in Figure 8. Second, there is the error termination due to overflow of arithmetic addition/subtraction or due to division by zero. Both situations are handled via pseudo-instructions that generate the break instruction, last rule in Figure 7.

We employ the reachability analysis, using the search command, for the state that has last as the current computational task. The timing information is updated along each execution path and, when we terminate the path exploration, the WCET is the maximum of these computed time values. Since we work on straight-line hard real-time programs, the state space exploration guarantees to terminate.

The actual search command that we use is: search initState =>* <T> <k> last </k> </T>, where initState is the initial state of the system. This state contains the program, represented as a term and loaded in the code memory cmem cell, the initialized register file regs cell, the instruction cache ic content with specified parameters param and reset profiling information profile, the data memory contents dmem, if initialized. Finally, and most importantly, the time cell, t with an initial time value of zero.

6 Conclusions

We present a method for WCET analysis of hard-real time programs, that is novel under two main aspects. With respect to the existing WCET estimation approaches, this is the first use of the formal executable semantics of an assembly language to derive useful abstractions. With respect to rewriting logic in general and K in particular, this is the first use of rewriting-based techniques for WCET analysis.

The concrete executable semantics is implemented in K-Maude, the K framework implementation on top of Maude system. This semantics is used to develop a very simple abstract semantics that enables all possible executions, via symbolic value for program variables. We use Maude’s search command, a reachability states exploration technique to compute the WCET estimation. The analysis is performed in the presence of instruction caches, that are also defined in K.
Our first priority is to improve the overall method for WCET analysis by improving the abstract semantics. A good abstract semantics helps to eliminate much of the infeasible execution paths and to lead to tight and accurate timing bounds. The first target is to rely on a value analysis to compute bounds for program variables. Our second target is to refine the current micro-architecture considerations and to integrate, in a modular way, K-based pipeline models, in short term a in-order and in long term an out-of-order pipeline. All these should conduct to the first WCET analyzer using the rewrite-based technology.

References

Certifying Execution Time

Nuno Gaspar\(^1\), Vítor Rodrigues\(^2\), Simão Melo de Sousa\(^1\)

\(^1\) RELiablE And SEcure Computation Group
Universidade da Beira Interior, Covilhã, Portugal
\(^2\) DCC-Faculty of Science, Universidade do Porto, Portugal
\(^3\) LIACC, Universidade do Porto, Portugal

Abstract. In real-time systems, timing constraints are essential to assure that all time-critical tasks meet their specified deadlines. In this context, the calculation of execution time bounds, both upper- and lower-bounds, is of paramount importance for schedulability analysis. Execution times depend both on the program flow and on complex microarchitectural features like caches and pipelines. Therefore, schedulability analysis of run-time programs must rely on safe control flow analysis methods and strongly depends on the accurate modeling of the underlying hardware.

In this paper we outline our framework for timing analysis of embedded real-time systems Abstraction-Carrying Code Platform for Timing validation (ACCEPT). We describe in detail the component CATA - CertificAtes for Timing Analysis, responsible for generating and checking execution time bounds certificates. In the context of mobile code safety, the validity of these checkable certificates entails compliance with the computed execution time bounds. The certificates generation/checking mechanism based on Abstraction-Carrying Code allows us to minimize the trusted computing base and provides a simple and a low resource consuming method for the checking phase.

Keywords: real-time systems, worst-case execution time, abstract interpretation, abstraction-carrying code, fixpoint computation, timing validation.

1 Introduction

Real-time systems can be seen as sets of concurrent tasks that are expected to execute programs subject to some predefined timing constraints. One way to perform schedulability analysis on these programs is to evaluate the correctness of their timing behavior based on the Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET) of each associated task. This type of information can only be obtained at machine-code level since it strongly depends on hardware factors. However, modern processors include mechanisms like caches and pipelines which make instruction’s execution time context dependent, increasing the difficulty of execution time estimation. To cope with this, one could be tempted to always assume the local worst case scenario (e.g. all cache memory accesses have a miss penalty associated) in order to obtain safe predictions.
Nonetheless, two problems may arise with such approach. Firstly, it may lead to excessive over-approximations of the actual WCET, which may lead to the over-dimensioning of hardware resources. Secondly, the obtained predictions may in fact be unsafe due to timing anomalies [28].

Mobile code safety has been progressively gaining notoriety in the sphere of real-time systems [29], both at research and industry levels. It represents an enabling technology for opening the real-time software development process to the mobile code paradigm (including upgrade and updates), tackling the limitations of standard client-server based approaches. Mobile code safety between two nodes in a network can be achieved by analyzing untrusted programs and producing associated certificates on the producer side, and by efficiently checking that the code+certificate package is valid on the receiver side.

One could argue that typically, applications loaded in embedded systems have no real-time requirements (e.g. ring tones for mobile phones). However, in [22], Kirsch et al identify the mobility of real-time programs as a challenging, yet desirable feature of embedded systems. Indeed, even a software/system update can be seen as mobile code. Thus, the ability to independently validate the timing behavior of programs would be of major interest for such systems. Moreover, a timing validation mechanism could also be a valuable asset for equipment manufacturers and software producers.

Approaches to this problem include Proof-Carrying Code [5] (PCC), Typed-Assembly Languages (TAL) [8] and Abstraction-Carrying Code (ACC) [2]. Our solution falls the third category as we propose the use of Abstract Interpretation [7] for static analysis and generation of certificates. Besides full automatic analysis, advantages of this approach are: (1) more flexibility in the choice of abstract properties; (2) inclusion of precision issues on the analysis; (3) use of low-consuming certificate checking mechanisms. ACC guarantees the simplicity to check the received certificate by generating the certificate as a fixpoint. On the consumer side, it is only necessary to check that the certificate is indeed a fixpoint within a single fixpoint iteration [2]. In addition, an useful feature of ACC is the possibility to compare the intended behavior, specified by safety assertions, with the computed behavior.

The determination of safe and precise bounds for execution time has been the object of intensive study in the literature [32]. In the context of mobile code safety, existing methodologies employ formal approaches that rely on a potential untrusted third party, without being able to check the specified time bounds in a stand-alone manner. In such a context, the computed execution time bounds are trusted and then not verified, despite mobile code can be tampered by the code communication. Thus, the safety requirements in embedded systems must consider potentially modified code.

The cache analysis in our framework is based on the theory of abstract interpretation and used to predict instruction cache hits and misses [11, 33, 31]. Then, the BCET and WCET bounds are calculated by means of Integer Linear Programming (ILP) techniques [25], which take as input the previous information together with the program flow information gathered at source-code level.
by means of user annotations. These annotations contain required information for tight time bounds estimation, such as loop bounds and infeasible paths, and also the specification of the intended timing behavior. Our main contributions when considering the overall structure of embedded real-time systems are:

- production and checking of timing certificates in the context of mobile code safety using stand-alone mechanisms;
- application of abstraction-carrying code techniques to the execution time certification of machine code by extending state of the art static analysis.

The remainder of this paper is organized as follows. In the next subsection we present the related work. In section 2 we briefly present the broader project in which CertificAtes for Timing Analysis (CATA) is included in, and then proceed by describing the fundamentals that underlie the tool itself (section 3). Building on top of these fundamentals, sections 4, 5 and 6 detail CATA. Experimental results and benchmarking are presented in section 7. Final remarks and directions for future work are discussed in section 8.

Related Work

There are numerous approaches in the literature focused in algorithms and tools for the computation of execution times [32, 3, 10, 24]. Our goal here is not to present yet another execution time framework, but rather to emphasize the process of production of certificates using a static analyzer that predicts execution time bounds, and the corresponding verification process that assures the validity of certificates.

Proof-Carrying Code (PCC) [27] is a general mechanism based on theorem proving to enable a program consumer to locally check the validity of the code w.r.t. some safety policy. It is based on automatic generation of certificates that constitutes a proof that the program is correct and an efficient certificate checking mechanism on the program consumer side. The key benefit of this approach is that there is no need to trust any third party.

The problem of certifying resource consumption, namely execution times, has already been addressed before. In [9], Crary & Weirich propose an extended type system capable of specifying and certifying bounds on resource consumption, albeit without determining bounds on execution times. The result of their approach is an executable that is certified w.r.t. resource consumption.

Furthermore, Bonenfant et al and Jost et al [6, 21] present an interesting combination of information retrieved at source code level, with low-level timing information gathered with the aiT tool [11]. Both works provide guaranteed bounds on worst-case execution times for a strict, higher-order functional programming language.

The Mobility, Ubiquity and Security (MOBIUS) [4], and the Mobile Resource Guarantees (MRG) [17] research projects also aim at the certification of resource consumption. Their approaches rely mostly on theorem proving, whereas our framework has its foundations on the theory abstract interpretation.
Kwon et al present in [23] a safe mobile code representation for Real-Time Java programs. Based on a Static Single Assignment (SSA) representation, it is argued that this approach is able to cope with the intrinsic challenges of high-integrity real-time software, such as subsystems upgrades and software portability.

2 Proposed Architecture

The ACCEPT platform is generically described by Figure 1. On the supplier side, we consider a Design-by-Contract [19] approach where a high-level programming language is extended with annotations that define the intended timing properties for each procedure. By placing these annotations directly into the source code, we are able to express assertions on the results produced by the subsequent timing analysis. The annotation mechanism also provides useful information for CATA such as loop bounds and infeasible paths. This is achieved through a timing analysis-aware compilation process preserving the annotations semantics.

Fig. 1. ACCEPT: Abstraction-Carrying CodE Platform for Timing validation

In one hand, an accurate timing analysis is only possible at hardware level. On the other hand, feedback about the compliance of the timing specification must be done at source-code level. Hence, in order to perform timing validation w.r.t. the timing specifications, timing analysis is performed at machine-code level, taking into account the specific features of the processor environment. Afterwards, any possible non-compliance of the computed execution time bounds with the timing specification is signaled to the source-code level with an alert my means of back annotations [20]. The idea of this mechanism is to propagate the timing information back to the source-code level, warning the system developers about violations w.r.t. the timing specification. As a side note, this back-annotation mechanism is very similar to the process used in the Frama-C framework [12], where proof obligations are associated with the code snippets which originated them.

The safety in terms of real-time constraints in mobile embedded code systems is done in two phases, both on supplier and consumer sides. The first phase involves generation/checking of certificates and the second phase deals with calculation/verification of a set of linear equation systems in ILP.

On the supplier side, the certificate is produced by CATA in the form of a fixpoint. The principle of the analysis is to compute an approximate semantics of
the program using abstract interpretation. The abstract evaluation of a program computes by successive approximations abstract invariants associated to every program point. To this end, microarchitectural analysis [31] is performed for each component in the processor environment that may affect timing behavior and for all possible program paths. Then, for WCET a posteriori path analysis finds the longest path in the control-flow graph using the information produced by the microarchitectural analysis. This is achieved by maximizing an ILP problem. This process is performed analogously for BCET.

On the receiver side, the certificate validation uses the same static analyzer that produced the certificate. A certificate is considered valid if the fixpoint solver is able to assure that one simple one-pass iteration does not affect the fixpoint. This is a fully automatic and stand-alone process that successfully provides the required low-consuming checking mechanisms of mobile code safety. Afterwards, the consumer can compute the execution time bounds by the same ILP techniques.

3 Fundamentals

Abstract interpretation is a semantics-based theoretical framework that copes with undecidability and complexity using approximations of sets and set operations. The soundness of an abstract interpretation is proved with respect to a particular program semantics called collecting semantics. For this purpose, a formal description of the standard semantics describing the possible behaviors of programs is required in order to gather information about all program executions. However, the state space considered by the collection semantics is usually too large for an exhaustive execution of all possible program sequences. Thus, in order to effectively compute properties about programs, an abstract semantics is used. The soundness of this abstract semantics relatively to the collection semantics is given by the existence of a Galois connection between an abstract domain $D^\sharp$ and a concrete domain $D$, respectively.

The collection semantics of a program $P$ is defined as the fixpoint of a set of recursive equations [33] over the domain of concrete values $D$. The abstract interpretation of a program $P$ is performed by interpretation $P$ over a simpler abstract domain $D^\sharp$. This abstract domain allows to trade efficiency over precision, that is, the fixpoint over the abstract domain is an over-approximation, yet a sound approximation, of the collecting semantics.

The framework of abstract interpretation not only provides an adequate framework to reason about fixpoints, but also an elegant way to infer abstract models of programs that can play the role of certificates. In this sense, certificates are defined as fixpoint abstract semantics. This approach was first used in the context of mobile code safety by the Abstraction-Carrying Code (ACC) framework [2]. The most innovating aspect of this approach is the definition of a fixpoint checker that is able to validate certificates using a simple one-pass fixpoint iteration, therefore meeting the low resource consuming requirement of mobile code systems.
Next we describe how program semantics are expressed in abstract interpretation. A program is characterized by its control-flow graph, constituted by a set of edges \( E \subseteq V \times \text{Ins} \times V \), where \( V \) represents the program points, \( v_i \in V \) models the program’s entry point and \( \text{Ins} \) models the instruction associated with edge \( E \). A standard semantic function \( J : \text{Ins} \to (S \to S) \) assigns to each \( \text{ins} \in \text{Ins} \), a transfer function that models its effect on the program state \( S \) being evaluated.

The collecting semantics \( CS : V \to \wp(S) \) assigns for each program point \( V \), the set of program states \( S \) that may occur in any possible execution (where \( \wp(S) \) stands for powerset of \( S \)).

The static analyzer computes the fixpoint of a finite number of equations extracted from the control flow graph of the program. There are two types of equations: the first relates exit with entry information for every program point \( V \) and the second relates entry information of a program point \( V_i \) with exit information of nodes from which there exists an edge to the program point \( V_i \), i.e., \( \bigcup \{ V_j \mid (V_j, \text{ins}, V_i) \in E \} \).

The system of recursive equations is solved by computing the least fixpoint \( \text{lfp}(F) = F^n(\lambda v. \emptyset) \) of the functional \( F : (V \to \wp(S)) \to (V \to \wp(S)) \):

\[
F(f)(v') = \begin{cases} 
S_0 & \text{if } v' = v_{\text{in}}, \\
\bigcup_{(v, \text{ins}, v') \in E} \{ \text{ins} \}(f(v)) & \text{otherwise},
\end{cases}
\]

where \( S_0 \subseteq S \) is the set of initial states.

However, as mentioned in Section 2, the collection semantics is usually not efficiently computable for large sets. For this reason, the analysis is performed on a simpler abstract domain \( D^\sharp = (S, L, \beta, \gamma) \), where \( L = (L, \sqsubseteq, \sqcup, \bot, \top) \) is a complete semi-lattice and \( \beta : S \to L \) is a representation function that maps concrete states to abstract states. The idea, is that \( \beta \) maps a state \( S \) to the best property describing it. Conversely, \( \gamma : L \to \wp(S) \) is a concretization function that maps abstract states to concrete states.

As mentioned above, the collecting semantics operates over sets of states, while the abstract semantics operates over the domain of properties. The objective is to define an adjunction between the concrete and abstract domains so that a concretization followed by abstraction do not incur in loss of precision. The abstraction function \( \alpha \) is a complete join morphism and can be defined using the representation function \( \beta : \wp(S) \to L \), by \( \alpha(S') = \bigsqcup \{ \beta(s) \mid s \in S' \} \). The concretization function \( \gamma \) and the abstraction function \( \alpha \) define the Galois connection

\[
\langle \wp(S), \subseteq \rangle \xrightleftharpoons{\alpha, \gamma} \langle L, \sqsubseteq \rangle
\]

such that \( \alpha(X) \subseteq l \iff X \subseteq \gamma(l) \). Furthermore, in order to ensure termination, it is required that the Ascending Chain Condition holds, i.e., that every ascending chain of elements eventually terminates. To this end, both abstraction and concretization functions must be monotonic w.r.t. the \( \subseteq \) and \( \sqsubseteq \) operators, respectively. Then, fixpoint over-approximation is defined as follows: given the continuous functional \( F \), the standard fixpoint transfer theorem says that if one
take a correct approximation \( F^{♯} \sqsubseteq \alpha \circ F \circ \gamma \) in \( D^{♯} \), then

\[
lfp(F^{♯}) \sqsubseteq \alpha(lfp(F))
\]

where \( lfp(F^{♯}) \) in an over-approximation of \( lfp(F) \) in the sense that \( \gamma(lfp(F^{♯})) \sqsubseteq lfp(F) \). The semantic function \([\cdot]^{♯} : \text{Ins} \rightarrow (L \rightarrow L)\). Then, the system of recursive equations in the abstract domain is solved by computing the fixpoint \( lfp(F^{♯}) = F^{♯n}(\lambda v.\bot) \) of the functional \( F^{♯} : (V \rightarrow L) \rightarrow (V \rightarrow L) \):

\[
F^{♯}(f)(v') = \begin{cases} 
l_0 & \text{if } v' = v_{\text{in}}, \\
\bigcup_{(v,\text{ins},v') \in E} \left[\text{ins}\right]^{♯}(f(v)) & \text{otherwise},
\end{cases}
\]

where the abstraction of the concrete initial states is defined as the initial abstract state, thus \( \alpha(S_0) \sqsubseteq l_0 \).

Required that the abstract transfer function \([\text{ins}]^{♯}\) is monotonic w.r.t. the \( \sqsubseteq \) operator, we obtain by induction that \( F^{♯n}(\lambda v.\bot) \sqsubseteq F^{♯(n+1)}(\lambda v.\bot) \) for all \( n \).

All the elements of the sequence are in \( L \), and since this is a finite set, not all elements of the sequence can be distinct. Thus, there must be some \( n \) such that:

\[
F^{♯(n+1)}(\lambda v.\bot) = F^{♯n}(\lambda v.\bot)
\]

The solution to this equation system is designated as the least fixed-point, \( lfp(F^{♯}) \):

\[
F^{♯(n+1)}(\lambda v.\bot) = F^{♯n}(\lambda v.\bot) = lfp(F^{♯})
\]

Static analyses are obtained by means of instantiations of the fixpoint framework, where the specificities of a particular target processor are taken into consideration. Next, we present a static cache analysis tailored for the ARM920T processor.

## 4 Static Cache Analysis

Most modern processors feature cache mechanisms, being the ARM920T processor one example. Caches are used to improve memory access times. In fact, excluding registers, caches have the fastest access times of all storage systems. Therefore, they are of paramount importance for real-time systems, as they reduce the number of cycles spent while fetching data.

One of the aspects that characterize a cache is its associativity, i.e., the number of possible locations, each one identified by a tags, where a memory block can be present inside a cache block. Generically, in a A-way set associative cache, the tag address space is \( A \) and the cache is composed by a number of cache sets, each hold \( A \) ways.

Determining whether a memory access will be a definite hit or miss is of paramount importance for an accurate timing analysis. The idea is to compute
for each program point an approximation of the cache contents, thus being able to safely determine if a cache hit or miss will occur for a particular memory address. Typically, static cache analysis computes Must and May information, which can be used together in order to infer a tight but safe approximation of the concrete cache contents.

In [16], Grund and Reineke present a cache static analysis for the FIFO replacement policy. They claim that for a $A$-way cache set, and a memory access sequence containing $p \geq A$ pairwise different memory blocks, at least $p - A$ misses must happen. This is justified by the fact that only accessed memory blocks are inserted into the cache. However, as we will see in the following, this does not hold for the ARM920T.

The ARM920T can be setup with FIFO as replacement policy, and since each cache set behaves independently, only one cache set needs to be analyzed. Let $s$ denote a cache set, with the cache ways $l_i$ ordered from left to right, then we can model a FIFO cache set in the following manner:

$$s = [l_0, ..., l_{A-1}] \in S$$

The claim made in [16] is not sound as it assume that only accessed blocks are inserted in the cache, and this need not to be true. For instance, in the ARM920T whenever a cache miss occurs while fetching for an instruction, not only this instruction is inserted in the cache, but also the three subsequent ones. This occurs as each $l_i$ can hold up to four instructions. Thus, a cache update can be modeled as follows:

$$U([l_0, ..., l_{A-1}], m_j) = \begin{cases} [l_0, ..., l_{A-1}] & : \exists i : m_j \in l_i \\ [l_n, l_0, ..., l_{A-2}] & : \text{otherwise} \end{cases}$$

where $m_j$ is the memory block pointed by the program counter $j$ and $l_n = \{m_j, m_{j+1}, m_{j+2}, m_{j+3}\}$, that is, the new inserted contents.

The first case happens in the presence of a cache hit, i.e., no modifications are made. In case of a miss, the contents of the cache lines are shifted, and four new instructions are inserted.

This interesting feature makes the cache replacement policy more predictable, and therefore permits a better cache analysis. Even without a *May analysis* one can predict cache hits and therefore yielding a sharper analysis.

The abstract domain is defined as follows.

$$D^\sharp := [M_0, ..., M_{a-1}],$$

where $M_i$ is a set of memory blocks, and $a$ the number of ways for each cache set. The position of a memory block in the Must domain is an upper-bound on the number of misses that happened since its insertion.

Moreover, it is meaningless to allow different upper-bounds for the same memory block, therefore we constrain our domain with the following restriction

$$\forall i \neq j : M_i \cap M_j = \{\}. $$
Before moving to the abstract update function we need to define a classification function $C$. If the accessed memory block is contained in any of the $M_i$-sets, then it is a hit. If it is not the case, then being able to predict the access as a miss or unclassified depends on the number of cached memory blocks. If there are $a$ memory blocks in the cache, no other memory block may be cached, therefore we can predict a miss. The update function $U$ is defined in a similar way:

$$C : D^z \times M \rightarrow Class$$

$$C([M_0, ..., M_{a-1}], m) = \begin{cases} H : m \in \bigcup_i M_i \\ M : m \notin \bigcup_i M_i, |\bigcup_i M_i| = a \\ \bot : \text{otherwise} \end{cases}$$

$$U^z : Must \times M \times Class \rightarrow Must$$

$$U^z([M_0, ..., M_{a-1}], m_j, class) = \begin{cases} [M_0, ..., M_{a-1}] & : class = H \\ [M_n, M_0, ..., M_{a-2}] & : class = M \\ [{}, M_0, ..., M_{a-2} \cup M_{fresh}] & : \text{otherwise} \end{cases}$$

where $M_n = \{m_j, m_{j+1}, m_{j+2}, m_{j+3}\}$.

Naturally, depending on the classification the update function will behave accordingly. In case of a hit, the abstract domain is left unchanged. The two other cases only differ in the position where the memory block is inserted. In case of a miss, we can safely insert it in the first position of the cache, as we will be able to safely predict hits for it until a further misses occur. As for the last case, since we do not know for sure if it was a hit or miss, we can only safely predict hits for it until the next miss.

The join function is to be applied whenever we need to merge the result of more than one program path, i.e., program points with more than one incoming edge. These situations will often result in a loss of information, as we only can consider safe information. In this case, safe information means that we will only keep the memory blocks that are in both abstract cache sets.

$$J^z : D^z \times D^z \rightarrow D^z$$

$$J^z([A_0, ..., A_{a-1}], [B_0, ..., B_{a-1}]) := [C_0, ..., C_{a-1}]$$

where $C_k := \{m \in M : \exists i, j : m \in A_i \cap B_j, k = max\{i, j\}\}$. For the sake of clarity, consider the following example. Let $A = [\{f\}, \{\}, \{a, c\}, \{b\}]$ and $B = [\{\}, \{d\}, \{b, c\}, \{a\}]$. Their join would yield $J(A, B) = [\{\}, \{\}, \{c\}, \{a, b\}]$.

For instance, consider the program illustrated by the CFG in Figure 2. We consider a 4-way set associative cache, with cache invalidation at the beginning of the program’s execution. For simplicity, we constrain ourselves with only one cache set. We will start by defining the CFG’s abstract semantics, i.e., the abstract transfer functions.
The abstract semantics models the effects of the incoming edges to the current node onto the cache. Using Equation (2) of abstract transfer functions, we describe program behavior in terms of the program points, $X_1, X_2, \ldots, X_5$, the transfer function $U^\#: X_1, a$ and the join operation $J^\#$. Then, using Equation (3), we apply these abstract transfer functions until we reach a fixpoint. Since we assume cache invalidation, we start fixpoint iterations with an empty cache ($\bot$) at every program point. After fixpoint stabilization, defined by Equation (4), we reach the smallest fixpoint. This fixpoint in the abstract domain allow us to determine whether the memory accesses will hit or miss the cache, according to the classification function (6), and thus whether or not a cache miss penalty should be added to the execution of instruction at every program point.

Furthermore, in our framework we enhance predictability by unfolding the first iteration of the loops present in the program. Usually, the memory accesses of the first iteration will miss the cache, while the ones on the remaining iterations will hit. However, the loss of information due to the join of program paths will make this information disappear, and therefore no safe cache hits would be predicted. By unfolding the first iteration however, we are able to setup contexts for the analysis, and thus predict more hits. For more details regarding this process we refer the reader to [14].

5 Bound Calculation

The previous cache analysis determined execution time bounds on the basic blocks. Determining the WCET and BCET is now a matter of finding the paths that lead to the longest and shortest execution time, respectively. In timing analysis this is usually achieved by means of integer linear programming techniques.

Let $t_{w_{cet}}$ denote the longest execution time of a program. Computing $t_{w_{cet}}$ could be performed simply by summing the execution time of the $n$ nodes $v \in V$, belonging to the path leading to the longest execution time:

$$t_{w_{cet}} = \sum_{i=1}^{n} t(v_i),$$

where $t(v_i)$ is the execution time of node $v_i$. This could be feasible, if the path leading to the longest execution would be known beforehand. Determining it
can be too expensive for real programs. To overcome this obstacle, rather than analyzing paths explicitly, Implicit Path Enumeration [25] counts the number of executions of basic blocks. Hence, the above sum can be reformulated as:

\[ t_{wcet} = \sum_{v \in V} t(v_i).cnt(v_i), \]

where \( cnt(v_i) \) stands for the execution count of node \( v_i \). This sum is a linear combination, since the execution times \( t(v_i) \) are constants, as they have been computed in the previous cache analysis. Moreover, it is clear that the \( cnt(v_i) \)'s must be integer values. These are constrained according to the program’s behavior. There are two types of constraints: program structural constraints and program functionality constraints. The former constraints are derived automatically from the program’s control-flow graph, while the latter are either derived via a previous static analysis or (preferably) provided by the user, for instance, via annotations.

The number of times \((n)\) that program executions \((e_i)\) enter a node \(v\), is equal to the number of times \((m)\) that program executions \((e'_i)\) leave the same node, which is also equal to the node execution count, i.e. \(cnt(v)\). This principle is applied as a rationale to automatically obtain the structural constraints.

\[ \sum_{i=1}^{n} cr(e_i) = cnt(v) = \sum_{i=1}^{m} cr(e'_i) \]

6 Validation on the Consumer Side

In the context of mobile code safety consumers cannot trust the origin of the program due to potentially modified code attacks. Moreover, traditional methodologies for mobile code safety are based on a potentially untrusted third-party which increases the complexity and vulnerability of software update operations, as well as the trusted computing base of the overall system. On the contrary, our approach allows the consumer to verify the uploaded software in a standalone manner, thereby avoiding the need to trust an external certifying entity. The consumer locally checks the software timing behavior against the received certificate using efficient algorithms in terms of resource consumption.

For our purpose, we only need to check the received fixpoint w.r.t the performed static analysis. The verification process proceeds in three phases: the first phase checks the abstract invariants contained in the received fixpoint, which correspond to the execution times on each program point and for all possible program paths; the second checks a given set of assertions defined over the abstract semantics; and the third phase checks whether the global execution time received is a solution of a set of linear equations locally extracted from the uploaded software.

In order to check the compliance with the intended timing specification, the two first steps build the program’s control-flow graph and verify that the fixpoint contained in the certificate \((Cert_f)\) is a valid abstraction with respect to the
safety policy, that is, the set of assertions over execution times. The checking procedure can then be written as:

\[
\text{Check}(Cert_f) = \begin{cases} 
\text{True} & \text{if } F^\sharp(Cert_f) = Cert_f \land \text{assert}(Cert_f) = \text{True} \\
\text{False} & \text{otherwise}
\end{cases}
\]

where \( F^\sharp(Cert_f) \) is the computed fixpoint on the consumer side. The concept of safety in terms of execution times is briefly described in the following. By the fact that the certificate supposedly contains a fixpoint, another iteration over it cannot change any of the abstract invariants. Thus, on the consumer side, a simple one-pass computation is sufficient to check that the certificate is indeed a fixpoint. However, the existence of a fixpoint is not sufficient to ensure the timing specification. For example, consider the scenario in which a malicious entity is able to tamper the certificate with potentially harmful data, but still leaving the abstract invariants behaving as a fixpoint. Then, the set of assertions defined on the consumer side must detect when the executions times contained in the fixpoint are outside the timing specification. In this way, we guarantee that the uploaded software does not threat the mobile system, at least with respect to the timing behavior.

In the cases where the received certificate does not behave as a fixpoint, the program consumer can simply reject the program. One could argue that the program could run, and have its execution killed in the case of a non-compliance with the timing specification. However, that constitutes a waste of resources, and in the context of embedded systems it is unacceptable, due to very limited computational resources. On the other hand, if the certificate is indeed a fixpoint, then the program consumer can locally compute the execution time bounds, and thus check the compliance with the timing specification. Furthermore, it should be noted that in our framework, it is possible for the program consumer to define new timing policies, for instance, the tightening of timing constraints.

7 Experimental Results

In order to assess CATA we performed a few experiments with the The Mälardalen WCET Benchmarks [18]. With the exceptions of insertsort.c and nsichneu.c the considered programs were slightly modified.\(^4\)

Table 1 presents some preliminary results. The WCET and BCET values are given as clock cycles. The GNU ARM toolchain [15] was used to compile. A cache-miss penalty of 64 cycles was assumed. Here, our objective is not compare these results with ones produced by commercial tools, such as aiT [11], because the analysis of the cache behavior in itself is not sufficient to produce tight WCET/BCET estimations.

As for the certificates, Table 2 gives a rough estimate on the execution time\(^5\) the processes took. These experiments were performed on a MacBook Pro 2.4GHz Intel Core 2 Duo.

\(^4\) Function calls were removed.
\(^5\) Obtained via the time utility.
### Table 1. Execution Time Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source Code LOC LOC</th>
<th>ARM Assembly LOC LOC</th>
<th>WCET ((cycles))</th>
<th>BCET ((cycles))</th>
</tr>
</thead>
<tbody>
<tr>
<td>bs.c</td>
<td>60 108</td>
<td>3049 450</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crc.c</td>
<td>107 277</td>
<td>1017084 1187</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fdct.c</td>
<td>144 732</td>
<td>82131 603</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fibcall.c</td>
<td>17 50</td>
<td>11767 457</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertsort.c</td>
<td>40 112</td>
<td>7833 930</td>
<td></td>
<td></td>
</tr>
<tr>
<td>janne_complex.c</td>
<td>27 73</td>
<td>5685 384</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jfdctint.c</td>
<td>160 759</td>
<td>148984 616</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ns.c</td>
<td>400 1410</td>
<td>90245 308</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsichneu.c</td>
<td>2362 9053</td>
<td>351653 377</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 2. Certificate Benchmarks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bs.c</td>
<td>16053</td>
<td>0.028s 0.014s</td>
</tr>
<tr>
<td>crc.c</td>
<td>70656</td>
<td>0.066s 0.054s</td>
</tr>
<tr>
<td>fdct.c</td>
<td>247410</td>
<td>0.386s 0.337s</td>
</tr>
<tr>
<td>fibcall.c</td>
<td>10676</td>
<td>0.024s 0.011s</td>
</tr>
<tr>
<td>insertsort.c</td>
<td>33296</td>
<td>0.027s 0.018s</td>
</tr>
<tr>
<td>janne_complex.c</td>
<td>17251</td>
<td>0.032s 0.014s</td>
</tr>
<tr>
<td>jfdctint.c</td>
<td>295185</td>
<td>0.518s 0.496s</td>
</tr>
<tr>
<td>ns.c</td>
<td>45950</td>
<td>0.049s 0.032s</td>
</tr>
<tr>
<td>nsichneu.c</td>
<td>3424892</td>
<td>1m21.807s 1m16.892s</td>
</tr>
</tbody>
</table>

As expected, the time necessary to produce the certificate and the time necessary to solve the ILP problem is proportional to the size of the control flow graph. However, the observation of the time necessary to check the certificate shows that the checking phase is indeed faster, but not significantly faster when compared to the certificate production time. This is due to the fact that the ILP phase has to be recomputed in the consumer as well and behaves as the dominant function. Our current work is focused on the minimization of the time necessary for the ILP checking on the consumer side.

Further, the certificate is simply the dump of the fixpoint structure, and therefore may be considerably large, as is particularly notorious in the nsichneu.c benchmark. In the context of mobile code, the size of the certificate is also of extreme relevance. We are currently studying possible ways to reduce the number of labels in the control flow graph, so that smaller certificates can be produced without affecting the precision of the static analysis.

### 8 Conclusions and Future Work

Abstract interpretation has been widely used in the industry, being static timing analysis one of its most successful applications [33]. In our approach, we use abstract interpretation to obtain safe approximations on execution times, taking into account the hardware specificities.
Fixpoint approximation rules allow us to infer an abstract timing model of the program, which solution can then be used as a certificate on the produces side. Conversely, a program consumer can locally validate the received program w.r.t. to its timing behavior, by simply checking that this abstract model is indeed a fixpoint (a one-pass process), and then verify the execution time bounds with the received certificate.

At present there are still some open issues that remain to be addressed. One of the main challenges we face in order to make CATA useful in practice is the size of the certificates. Embedded systems are known for their scarce resources and limited computational means. With this in mind, Albert et al introduce in [1] the notion of a reduced certificate, with the objective to produce a certificate that only contains the essential information that the program consumer cannot reproduce by itself, without yielding an overhead in the certificate checking process.

Furthermore, with our current approach the consumer still needs to compute the execution time bounds with ILP. Indeed, the burden avoided by our fixpoint certificate methodology is only restricted to the program’s cache behavior, the same efficiency criteria should be applied to the ILP phase. There is clearly room for improvement. Our actual effort is focused on alternative methods that avoid to recompute the maximal (or minimal) solution of the ILP equation system.

Foreseen enhancements for our CATA framework include the integration of more static analyses. At the cost of increasing the size of the certificate, a pipeline analysis [30] would allow a sharper bound computation. Further, the ARM instruction set has the distinct feature of all instructions being suffixed by a conditional that needs to hold. This leaves room for the integration of a static analysis to determine the outcome of these predicates, and thus safely removing program paths whenever appropriate. In earlier work we presented one such analysis [13].

We are currently pursuing a pragmatical evaluation of our certificate generation component CATA. For now, we are not concerned with performance, but with the proof of concept and thus, obviously, correctness and adequacy in the context of mobile code.

To the best of our knowledge this is the first work applying the concepts of Abstraction-Carrying Code to the static timing analysis field.

Acknowledgments

This work is partially supported by the RESCUE project PTDC/EIA/65862/2006 funded by FCT (Fundaçãopara a Ciência e a Tecnologia) and by the FAVAS project PTDC/EIA-CCO/105034/2008, FCT. We also thank Jan Reineke for valuable discussions regarding the cache analysis and Bertrand Jeannet for troubleshooting with mllpsolve [26].

Available in the first author’s website http://webx.ubi.pt/~m2602/cata
References


Interpolation-based height analysis for improving a recurrence solver

Manuel Montenegro\textsuperscript{1}, Olha Shkaravska\textsuperscript{2},
Marko van Eekelen\textsuperscript{2,3}, and Ricardo Peña\textsuperscript{1}

\textsuperscript{1} Universidad Complutense Madrid
\textsuperscript{2} Radboud University Nijmegen
\textsuperscript{3} Open University of the Netherlands
\{montenegro\}@fdi.ucm.es,\{shkarav,marko\}@cs.ru.nl,\{ricardo@sip.ucm.es\}

Abstract. The COSTA system infers resource consumption bounds from Java bytecode using an internal recurrence solver PUBS. This paper suggests an improvement of the COSTA system, such that it can solve a larger number of recurrences. The idea is to replace one of its static analyses, the ranking function analysis, by another kind of analysis, height analysis, in such a way that polynomial bounds of any degree may be inferred instead of just linear expressions. The work can be seen as an application of some polynomial interpolation techniques used by some of the authors in prior analyses. Finding a way to choose proper test nodes is the key to the solution presented in this paper.

1 Introduction

The application of resource analysis techniques to actual programs frequently leads to generating cost recurrence equations which must be solved, i.e. expressed in a closed form in order to be useful. There have been many attempts to solve this kind of equations in an automatic way. Most of the tools doing this are restricted to some specific recurrence families, frequently on only one variable.

A system proved successful in generating and solving multivariate recurrence equations is COSTA \cite{1}, together with its associated recurrence solving subsystem PUBS \cite{2}. COSTA is given a program text written in Java bytecode, and the kind of resource to be analysed (number of instructions executed, memory consumption, and others), and in many cases it is able to obtain symbolic upper bounds for these runtime figures. The bounds are expressed as multivariate functions on the sizes of the input arguments of the method being analysed.

Internally, the system contains many different static analyses which we briefly summarise in Sec. 2. In this paper, we focus on one of them: the \textit{ranking function synthesis} done in one of the steps. Its aim is to obtain an upper bound on the number of unfoldings the recurrence equations must undergone to reach

This work was partly funded by the EU Artemis Joint Undertaking in the CHARTER project, grant-nr. 100039, and by Spanish FPU grant AP2006-02154 and projects TIN2008-06622-C03-01 (STAMP), S2009/TIC-1465 (PROMETIDOS).
a base case, i.e. to reach a case with does not admit more unfoldings. This number corresponds with the maximum height of the so-called evaluation trees. As we will see, this ranking function is crucial in the computation of the resource upper bound. The limitation of this step is that the ranking function must be a linear expression on the argument sizes. The Podelski and Rybalchenko’s method [19], which is known to be complete for linear ranking functions, is used to this purpose.

In the past, we have used polynomial interpolation techniques in different types of size analysis [20], and also in synthesising polynomial ranking functions for Java loops [22]. The main idea there was to synthesise polynomials by evaluating program fragments, or particular term rewriting systems, in some specific points forming a so-called NCA configuration (Node Configuration A, [7]). By adopting this configuration the number of test points needed is a precise function of the polynomial degree and the number of variables.

We apply the same idea here by evaluating recurrence equations in well chosen points and then interpolating a multivariate polynomial of known degree passing through those points. As we are only interested in the maximum height of the recurrence evaluation trees (the rest of the recurrence solving process is already done by PUBS), we first write a derived recurrence defining this height. The obtained polynomial would be then an upper bound of the height of any evaluation tree. The degree must be guessed, as it happens in other techniques inferring arbitrary polynomials [17, 13], and additionally the interpolated polynomial must be proved correct w.r.t. the given recurrence equations. We will show that the process can be performed fully algorithmically by using appropriate tools. In the end we have extended the power of COSTA by allowing solving more and more general recurrences, specifically those admitting a polynomial upper bound on the height of the evaluation trees.

The plan of the paper is as follows: After this introduction, in Sec. 2 we summarise the main components of the COSTA-PUBS system and the process it follows for recurrence solving; in Sec. 3 we explain the main steps of our inference process by using a small running example; sections 4, 5, and 6 contains the technical details of respectively what do we mean by evaluating a non-deterministic recurrence, how to choose appropriate test points, and how to prove the obtained polynomial correct; finally, Sec. 7 surveys the related work and draws some conclusions.

## 2 A Broad Overview of COSTA and PUBS

The COSTA System is based on the classical approach to resource analysis, due to Wegbreit [25]. Given a Java bytecode program and a cost model, COSTA generates, in a first phase, a set of equations specifying the cost of the program as a function on the size of its input. COSTA provides several notions of size which depend on the input’s type: The size of an integer is its value, whereas the size of an array is its number of elements. Finally, the size of an arbitrary object is the longest reachable pointer path that stems from that object. As an
example, we show below a code fragment with a recurrence relation specifying its memory consumption:

```java
while (n > 0) {
    int[] arr = new int[n];
    n--;
}
```

$$T(n) = \begin{cases} 0 & \{n \leq 0\} \\ 4n + T(n - 1) & \{n > 0\} \end{cases} \quad (1)$$

Although a recurrence relation captures precisely the cost of a program, an equivalent expression without recursion (closed form) gives a more intuitive idea about these costs from the programmer’s point-of-view. For instance, the recurrence shown above admits the following closed-form: $T(n) = 2n^2 + 2n$. So, the second phase of COSTA consists of the computation of a closed form for the previously generated set of equations. In contrast to already existing tools for solving recurrence relations [5], COSTA provides its own recurrence solver, PUBS [2]. The main reason is that, in practice, the set of equations capturing the cost of a program are not recurrence relations, but belong to the broader class of Cost Relation Systems (CRS), which are defined as follows:

**Definition 1 (Adapted from [2]).** A cost relation system is a finite set of mutually recursive equations of the form:

$$T(\bar{x}) = \exp + \sum_{j=1}^{l} D_j(\bar{y}_j) \psi$$

where $l \geq 0$, $\exp$ is a basic cost expression and $\psi$ is a set of linear relations on $\bar{x} \cup \{\bar{y}_j\}_{j=1..l} \cup \text{vars}(\exp)$ specifying the conditions under which the equation can be applied.

The definition of basic cost expressions is not relevant to this paper, and it shall be omitted here (see [2] for details). As pointed out by Albert et al. it is possible to unfold the definitions of the $D_j$ symbols in order to obtain equations of the form $T(\bar{x}) = \exp + \sum_{j=1}^{m} T(\bar{y}_j)$ for some $m \geq 0$. In the next sections we assume that this transformation has been done. The main difference between CRS and regular recurrence relations is non-determinism: when evaluating $T(\bar{v})$ for some concrete values, we might be able to apply more than one equation. Moreover, the arguments $T(\bar{y}_j)$ occurring in the recursive calls of an equation may not be uniquely determined by the guards, as in, for example, $T(x) = 1 + T(x') \{0 \leq x' < x\}$. Thus the evaluation of $T(\bar{v})$ may give rise to different results and hence a CRS defines a relation instead of a function.

The PUBS approach for computing a closed form upper-bound is based on the notion of an evaluation tree (ET), which is the structure that arises from evaluating $T(\bar{v})$ with a concrete vector value $\bar{v}$. Each node in the ET contains the cost of its basic cost expression $\exp$ applied to the $\bar{v}$, and its children correspond to the recursive calls $T(\bar{y}_j)$, where $j \in \{1..m\}$. Notice that the evaluation of $T(\bar{v})$ may give place to several ETs, because of non-determinism. The computation of an upper bound to $T(\bar{x})$ amounts to finding upper-bounds to:

97
The number of base and recursive nodes of every possible ET, respectively denoted by \( nb(\bar{x}) \) and \( nr(\bar{x}) \).

The value of the basic cost expressions occurring in the base and recursive nodes of every possible ET, respectively denoted by \( cb(\bar{x}) \) and \( cr(\bar{x}) \).

Given these, an upper bound \( T^+(\bar{x}) \) is computed as follows:

\[
T^+(\bar{x}) = nb(\bar{x}) \times cb(\bar{x}) + nr(\bar{x}) \times cr(\bar{x})
\]  

(2)

The computation of \( cb(\bar{x}) \) and \( cr(\bar{x}) \) is beyond the scope of this paper. With regard to the number of nodes in the ET (\( nb \) and \( nr \)), these functions can be approximated from the ET’s branching factor \( b \) and maximal height \( h(\bar{x}) \) as follows:

\[
nb(\bar{x}) = b^{h(\bar{x})} \\
nr(\bar{x}) = \begin{cases} 
h(\bar{x}) & \text{if } b = 1 \\
\frac{b^{h(\bar{x})} - 1}{b - 1} & \text{if } b > 1
\end{cases}
\]

The \( h \) function stands for the length of the maximal call chain (without including the base case) that stems from the root of the ET. For example, with the CRSs shown in (1) the only (and hence the longest) chain reachable from \( T(n) \) is as follows:

\[
\begin{align*}
T(n) & \rightarrow T(n-1) \rightarrow T(n-2) \rightarrow \ldots \rightarrow T(1) \rightarrow T(0) \\
h(n) & = n
\end{align*}
\]

In order to compute \( h(\bar{x}) \), PUBS derives a ranking function for \( T \) by applying Podelski and Rybalchenko’s method [19]. Unfortunately, this method fails when the ranking function does not depend linearly on the arguments \( \bar{x} \), as the following example shows:

**Example 1.** Let us assume the following loop:

```java
while(x > 0 || y > 0) {
    byte[] b = new byte[x];
    if (y == 0) { x--; y=x; } else { y--; }
}
```

If we consider memory consumption, COSTA would obtain the following CRS:

\[
\begin{align*}
T(x, y) &= \text{nat}(x) & \{ x = 0, y = 0 \} \\
T(x, y) &= \text{nat}(x) + T(x-1, x-1) & \{ x > 0, y = 0 \} \\
T(x, y) &= \text{nat}(x) + T(x, y-1) & \{ x \geq 0, y > 0 \}
\end{align*}
\]

where \( \text{nat}(x) \) abbreviates \( \max\{0, x\} \). This CRS cannot be solved by PUBS, since the length of the longest call chain depends on \( (x, y) \) in a non-linear way.

\[
\begin{align*}
T(x, y) & \rightarrow T(x, y-1) \rightarrow \ldots \rightarrow T(x, 0) \rightarrow T(x-1, x-1) \rightarrow \ldots \rightarrow T(0, 0) \rightarrow T(0) \\
h(x, y) & = \frac{1}{2}x^2 + \frac{1}{2}x + y
\end{align*}
\]
3 Interpolation-based call chain height analysis

COSTA derives cost recurrence equations from Java bytecode. From these CRSs COSTA derives ranking functions. Ranking functions are used to bound the height of the evaluation trees. In COSTA such ranking functions are limited to linear expressions. It is our goal to improve the approximation of the height of the evaluation trees by considering general polynomial expressions. Using polynomial interpolation we aim to derive those polynomial expressions by analysing the call chain directly. To this purpose we first derive equations for the height of the call chain from the COSTA cost relation equations. Then, we derive upper bounds by polynomial interpolation.

Firstly, we must determine the function to be interpolated. We transform a given original CRS in order to obtain a recursive definition modeling the height $T_h(\bar{x})$ of the evaluation tree:

- for non-recursive equations of the form:
  \[ T(\bar{x}) = \exp \psi \quad \text{we get:} \quad T_h(\bar{x}) = 0 \quad \psi \]

- for recursive equations of the form:
  \[ T(\bar{x}) = \exp + \sum_{i=1}^{m} T(\bar{y}_i) \psi \quad \text{we get:} \quad T_h(\bar{x}) = 1 + \max_{i=1...m} \{ T_h(\bar{y}_i) \} \quad \psi \]

The function $T_h$ is multivalued: $T_h(\bar{x})$ is the collection of the lengths of all the call chains rooting in $\bar{x}$. Recall, that the function $h(\bar{x})$ is the maximum length of all possible call chains from $\bar{x}$, therefore $h(\bar{x})$ is the strict upper bound for $T_h(\bar{x})$.

Next, after we have the recurrence relations for $T_h(\bar{x})$, our aim is to find a polynomial $T_h^+(\bar{x})$ such that $T_h^+(\bar{x}) \geq h(\bar{x})$. If $h(\bar{x})$ is a polynomial, then $T_h^+(\bar{x})$ may be found by the standard interpolation. For instance, this is the case for the example 1. We will consider it in more detail later in this section. If $h(\bar{x})$ is not a polynomial then the standard interpolation must be adjusted. In both cases the idea is to generate a candidate for $T_h^+(\bar{x})$ based on its values in some finite collection of nodes. Then the candidate is checked. Roughly, checking is done by substituting it into the recurrence relation for $T_h(\bar{x})$.

In this paper we consider in more detail the issues related to the generating a candidate for an upper bound. The most important issue here is to find a collection of test nodes such that the corresponding interpolation problem have the solution (and it must be unique). There are many ways to obtain test nodes. In our work we use the gradient-based approach, which is described in Sec. 5.

Now we need to recapitulate some prerequisites from the classical interpolation theory that are necessary to understand the major challenge: finding test nodes. Recall that a polynomial of degree $d$ and dimension $s$ (the number of variables) has $N_d^s = \binom{d+s}{s}$ coefficients. Let a set of values $f_i$ of a real function $f$ be given. A set $W = \{ \bar{w}_i : i = 1, \ldots, N_d^s \}$ of points in a real $s$-dimensional space forms the set of interpolation nodes if there is a unique polynomial $p(\bar{z}) = \Sigma_{0 \leq |j| \leq d} a_j \bar{z}^j$ with the total degree $d$ with the property $p(\bar{w}_i) = f_i$, 

where $1 \leq i \leq N_d^s$. In this case one says that the polynomial $p$ interpolates the function $f$ at the nodes $\tilde{w}_i$. The polynomial interpolation exists and is unique under some conditions on the data, which are explored in polynomial interpolation theory [7]. For 1-variable interpolation this condition is well-known: all the test-nodes must be different. We have a closer look at 1-variable interpolation to provide the reader with the intuition behind the general case. A polynomial $p(z)$ of degree $d$ with coefficients $a_1, \ldots, a_{d+1}$ can be written as follows:

$$a_1 + a_2 z + \ldots + a_{d+1} z^d = p(z)$$

The values of the polynomial function in any pairwise different $d + 1$ points determine a system of linear equations w.r.t. the polynomial coefficients. More specifically, given the set $(z_i, p(z_i))$ of pairs of numbers, where $1 \leq i \leq d + 1$, and coefficients $a_1, \ldots, a_{d+1}$, the set of equations can be represented in the following matrix form, where only the $a_i$ are unknown:

$$
\begin{pmatrix}
1 & z_1 & \cdots & z_1^{d-1} & z_1^d \\
1 & z_2 & \cdots & z_2^{d-1} & z_2^d \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
1 & z_d & \cdots & z_d^{d-1} & z_d^d \\
1 & z_{d+1} & \cdots & z_{d+1}^{d-1} & z_{d+1}^d
\end{pmatrix}
\begin{pmatrix}
a_1 \\
a_2 \\
\vdots \\
a_d \\
a_{d+1}
\end{pmatrix}
=
\begin{pmatrix}
p(z_1) \\
p(z_2) \\
\vdots \\
p(z_d) \\
p(z_{d+1})
\end{pmatrix}
$$

The determinant of the left matrix, which contains the measurement points, is called the Vandermonde determinant. For pairwise different points $z_1, \ldots, z_{d+1}$ it is non-zero. This means that, as long as the output size is measured for $d + 1$ different input sizes, there exists a unique solution for the system of equations and, thus, a unique interpolating polynomial.

The condition under which there exists a unique polynomial that interpolates multivariate data is not trivial. This condition on $W$ is geometrical: it describes a configuration, called Node Configuration A (NCA) [7], in which the points from $W$ should be placed in $\mathbb{R}^s$. The multivariate Vandermonde determinant computed from such points is non-zero. Thus, the corresponding system of linear equations w.r.t. the polynomial’s coefficients has a unique solution. For a two-dimensional polynomial of degree $d$, the condition on the nodes that guarantees a unique polynomial interpolation is as follows:

$N_d^2$ nodes forming a set $W \subset \mathbb{R}^2$ lie in a 2-dimensional NCA if there exist lines $\gamma_1, \ldots, \gamma_{d+1}$ in the space $\mathbb{R}^2$, such that $d + 1$ nodes of $W$ lie on $\gamma_{d+1}$ and $d$ nodes of $W$ lie on $\gamma_d \setminus \gamma_{d+1}$, ..., and finally 1 node of $W$ lies on $\gamma_1 \setminus (\gamma_2 \cup \ldots \cup \gamma_{d+1})$.

A simple example of NCA is given by a rectangular grid: there are $d + 1$ parallel lines, such that some $d + 1$ points lie on one line, $d$ points lie on another one, $d - 1$ points belong on some third line, etc.

The NCA configuration for $s$ variables (s-dimensional space) is defined inductively on $s$ [7]. Let $\{\tilde{z}_1, \ldots, \tilde{z}_{N_d^2}\}$ be a set of distinct points in $\mathbb{R}^s$ such that there exist $d + 1$ hyperplanes $K_j^s$, $0 \leq j \leq d$ with

$$\tilde{z}_{N_d^2-1+1}, \ldots, \tilde{z}_{N_d^2} \in K^d_s$$

$$\tilde{z}_{N_{j-1}^d+1}, \ldots, \tilde{z}_{N_d^2} \in K^s_j \setminus \{K^s_{j+1} \cup \ldots \cup K^s_d\}, \text{ for } 0 \leq j \leq d - 1$$

100
and each of set of points $\bar{z}_{N_{j-1}^s+1}, \ldots, \bar{z}_{N_j^s}$, $0 \leq j \leq s$, considered as points in $\mathcal{R}^{s-1}$ satisfies NCA in $\mathcal{R}^{s-1}$.

### 3.1 Application of the approach to an example

Recall the recurrence relations in the example 1. We have to find an interpolating polynomial of degree 2 for the corresponding call-tree-height function:

\[
T_h(x, y) = \begin{cases} 
0 & \{x = 0, y = 0\} \\
1 + T_h(x-1, x-1) & \{x \geq 1, y = 0\} \\
1 + T_h(x, y-1) & \{x \geq 0, y \geq 1\}
\end{cases}
\]

Evaluating $T_h$ in 6 points forming an NCA yields the following table:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>$T_h(x, y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$T_h(1, 1) = 1 + T_h(1, 0) = 1 + 1 + T_h(0, 0) = 2$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$T_h(1, 2) = 1 + T_h(1, 1) = 3$</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>$T_h(1, 3) = 1 + T_h(1, 2) = 4$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$T_h(2, 1) = 1 + T_h(2, 0) = 1 + 1 + T_h(1, 1) = 4$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$T_h(2, 2) = 1 + T_h(2, 1) = 5$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>$T_h(3, 1) = 1 + T_h(3, 0) = 1 + 1 + T_h(2, 2) = 7$</td>
</tr>
</tbody>
</table>

By solving the resulting system we get the polynomial $T_h^+(x, y) = \frac{1}{2}x^2 + \frac{1}{2}x + y$. It is a routine to check that it fits the recurrence relations. We compute the closed form expression for $T^+$ by using (2):

\[
T^+(x, y) = \text{nat}\left(\frac{1}{2}x^2 + \frac{1}{2}x + y\right) \ast \text{nat}(x) + \text{nat}(x)
\]

and taking into account that the branching factor in this case is $b = 1$. □

### 3.2 When standard interpolation needs adjustment

When $h(\bar{x})$ is not a polynomial, but still, bounded by some polynomial, the standard interpolation must be adjusted. This is a topic of the ongoing research.

In particular, taking into account that $T_h^+(\bar{x}) \geq h(\bar{x})$, the interpolation problem may be generalised by adding variables $\delta_{ij}$ to the system of linear equations for the coefficients of $T_h^+(\bar{x})$ so that the corresponding conditions in the nodes look like $T_h^+(\bar{x}_{ij}) + \delta_{ij} = h(\bar{x}_{ij})$. The amount of rows of the system should increase correspondingly, that is we need more test nodes. If by solving the system we obtain non-negative $\delta_{ij}$, then we have obtained a good candidate for $T_h^+(\bar{x})$.

We study other possibilities to adjust standard interpolation for finding $T_h^+(\bar{x})$ such that $T_h^+(\bar{x}) \geq h(\bar{x})$, like considering in-equalities $T_h^+(\bar{x}_i) \geq h(\bar{x}_i)$ and minimising an appropriate objective function over the unknown coefficients of the polynomial $T_h^+(\bar{x})$.  

101
3.3 Towards non-deterministic cost relation systems

When the CRS defining $T_h$ is deterministic (as in our previous examples) it defines a single-valued function whose evaluation can be done in the usual way, e.g. by unfolding. If we want to compute a upper-bound to $h(\bar{x})$ we just choose a set of points in the domain of $T_h$ lying in a NCA configuration, and evaluate $T_h$ in these points, as shown in the examples above.

However, the computation of an upper-bound in non-deterministic CRSs is far more involved. The main reason is that we cannot just choose a point $\bar{x}$ and obtain all the possible results of the evaluation of $T_h(\bar{x})$, since there could be infinitely many ETs resulting from it. In the next two sections we will explain how to obtain testing nodes in which the value of $T_h$ is known, and how to perform a search on those nodes in which interpolation is more likely to result in a correct upper-bound (gradient-based approach).

4 Evaluation of Cost Relation Systems

Before dealing with the evaluation of a CRS we have to define its semantics. By the sake of simplicity, we consider only CRSs with a single recursive call (the extension to CRSs with several calls is straightforward):

$$T_h(\bar{x}) = 0$$
$$T_h(\bar{x}) = 1 + T_h(\bar{x}')$$

(3)

These equations define a relation $T_h \subseteq N^s \times N_\infty$, where $N_\infty \eqdef N \cup \{+\infty\}$. The pair $(\bar{x}, n)$ belongs to $T_h$ iff $n$ is a result of the evaluation of $T_h(\bar{x})$. The intuitive meaning of $(\bar{x}, +\infty)$ being in $T_h$ is that the evaluation of $T_h(\bar{x})$ may lead to an infinite call chain. The ordering $\leq$ on natural numbers and the $+$ operator is extended to $N_\infty$ as usual. Given these conventions, the following definition provides a way to characterize the relation $T_h$ from the set of equations in (3).

**Definition 2.** The relation $T_h$ defined by the CRS in (3) is the greatest fixed point of the function $F : \mathcal{P}(N^s \times N_\infty) \to \mathcal{P}(N^s \times N_\infty)$, defined as follows:

$$F(X) = \{(\bar{x}, 0) \mid \psi_b(\bar{x})\} \cup \{(\bar{x}, n + 1) \mid \psi_r(\bar{x}, \bar{x}') \land (\bar{x}', n) \in X \text{ for some } \bar{x}' \in N^s, n \in N_\infty\}$$

We write $T_h = \text{gfp } F$.

As a notational convention, we consider relations $T_h \subseteq N^s \times N_\infty$ to be multivalued functions $T_h : N^s \to \mathcal{P}(N_\infty)$. Their domain, denoted by $\text{dom } T_h$, is the set of $\bar{x} \in N^s$ such that $T_h(\bar{x}) \neq \emptyset$.

**Example 2.** The following CRS

$$T_h(x) = 0 \quad \{x = 0\}$$
$$T_h(x) = 1 + T_h(x') \quad \{x > 0 \land x' < x\}$$

defines the following relation

$$T_h = \{0 \mapsto \{0\}, 1 \mapsto \{1\}, 2 \mapsto \{1, 2\}, \ldots, i \mapsto \{1..i\}, \ldots\}$$
Example 3. Consider the following example:

\[
\begin{align*}
T_h(x) &= 0 \quad \{x = 0\} \\
T_h(x) &= 1 + T_h(x') \quad \{x' > x\}
\end{align*}
\]

Let us prove that this CRS defines the following relation:

\[T_h = \{(0,0), (0, +\infty), (1, +\infty), (2, +\infty), \ldots\}\]

The operator \(F\), applied to this particular case, is defined as follows:

\[
F(X) = \{(0,0)\} \cup \{(x, n+1) \mid x < x' \land (x', n) \in X \text{ for some } x' \in \mathbb{N}, n \in \mathbb{N}_\infty\}
\]

It is easy to see that \(F(T_h) = T_h\). Hence, \(T_h\) is a fixed point. Now we prove that is the greatest one by contradiction: assume that there exists a \(T'_h \supset T_h\) such that \(T'_h = F(T'_h)\). Since \(T'_h\) strictly extends \(T_h\), we have two possibilities:

- \((x,0) \in T'_h\) for some \(x \neq 0\). This cannot happen, since \(T'_h = F(T'_h)\) and the only tuple that \(F\) can return with a 0 in its right-hand side is \((0,0)\).
- \((x,n) \in T'_h\) for some \(x \geq 0\) and some \(n\) different from 0 and \(+\infty\). Then, there must exist some \(x_1 > x\) such that \((x_1, n-1) \in T'_h\), which leads to a contradiction if \(n = 1\), as we have seen in the previous point. If \(n > 1\), and since \((x_1, n-1) \in T'_h\), there exists another \(x_2 > x_1\) such that \((x_2, n-2) \in T'_h\) and we apply the same reasoning as before. Eventually we will reach a tuple \((x_n,0) \in T'_h\) for some \(x_n > 0\), leading to a contradiction.

Therefore, the set \(T_h\) shown above is the relation defined by this CRS. This relation can be given as the following multivalued function:

\[
T_h(x) = \begin{cases} 
\{0, +\infty\} & \text{if } x = 0 \\
\{+\infty\} & \text{otherwise}
\end{cases}
\]

In order to apply the techniques explained in Sec. 3, it is necessary to choose a set of points and determine the maximum value returned by \(T_h\) when applied to each of these points. However, in general, it may be difficult to compute \(\max T_h(\vec{x})\) for an arbitrary \(\vec{x}\), due to the non-determinism of CRSs. In particular, there may be a possibly infinite amount of vectors \(\vec{x}' \in \mathbb{N}^n\) satisfying the recursive guard \(\psi_r(\vec{x}, \vec{x}')\), and hence being eligible to be passed as argument to the recursive call to \(T_h\).

Example 4. Assume the following CRS:

\[
\begin{align*}
T_h(x) &= 0 \quad \{x \geq 100\} \\
T_h(x) &= 1 + T(x') \quad \{0 \leq x < 100, x < x'\}
\end{align*}
\]

We get \(T_h(0) = \{1..100\}\), but there are infinitely many ways of deriving \((0,1) \in T_h\). In general, for any \(x' \geq 100\) we obtain \((x',0) \in T_h\) and hence \((0,1) \in T_h\).
Given these difficulties, we will consider the evaluation of $T_h$ in a bottom-up fashion: we start from the set of points $A_0$ such that the evaluation of $T_h$ returns $\{0\}$. These points are known because they satisfy the base guard, but not the recursive one. In the next step, we consider the set of points $A_1$ that satisfy the recursive guard, but the corresponding recursive call falls into a base case. In general, our aim is to find a hierarchy of sets $A_0 \subseteq A_1 \subseteq \ldots \subseteq A_i$, where each $A_i$ contains the values of $x$ such that the evaluation of $T_h(x)$ does not require more than $i$ steps.

**Definition 3.** Given a relation $T_h : \mathbb{N}^* \to \mathcal{P}(\mathbb{N})$ and $i \in \mathbb{N}$, we define the set $A_i$ as follows:

$$A_i = \{ \bar{x} \in \text{dom } T_h | \max T_h(\bar{x}) \leq i \}$$

**Example 5.** Back to our Example 2, we obtain $A_i = \{0..i\}$ for each $i \in \mathbb{N}$, whereas in Example 3 we get $A_i = \{0\}$ for each $i \in \mathbb{N}$.

Our next step is to find a characterization of these $A_i$ sets in terms of the guards occurring in the CRS. This characterization is given as a set of predicates $\varphi_i$, defined as follows:

**Definition 4.** Given the CRS in (3) and for each $i \in \mathbb{N}$, we define the predicate $\varphi_i$ as follows:

$$\varphi_0(\bar{x}) \overset{\text{def}}{=} \psi_b(\bar{x}) \land \forall \bar{x}'. \neg \psi_r(\bar{x}, \bar{x}')$$

$$\varphi_i(\bar{x}) \overset{\text{def}}{=} \varphi_0(\bar{x}) \lor \left( (\exists \bar{x}'. \psi_r(\bar{x}, \bar{x}')) \land \forall \bar{x}'. (\psi_r(\bar{x}, \bar{x}') \Rightarrow \varphi_{i-1}(\bar{x}')) \right)$$

where $i > 0$

By using quantifier elimination methods we can express every $\varphi_i$ as a finite union of convex polyhedra. In [18] a survey of quantifier elimination techniques can be found. Most of these methods only apply to dense linear orders [26]. Cooper [9] developed a quantifier elimination procedure for Presburger arithmetic extended with a divisibility predicate. This is the method we are going to use in the subsequent examples. In our implementation we rely on a computer algebra system and its extension to a computer logic system [12, 10].

**Example 6.** We get the following predicates from the CRS given in Example 2:

$$\varphi_0(x) \equiv x = 0 \land \neg \exists x'. (x > 0 \land x' < x)$$

$$\equiv x = 0 \land \neg (x > 0) \equiv x = 0 \{\text{quantifier elimination}\}$$

$$\varphi_1(x) \equiv x = 0 \lor (x > 0 \land \forall x'. [x > 0 \land x' < x \Rightarrow x' = 0])$$

$$\equiv x = 0 \lor (x > 0 \land \neg [x > 0 \land 1 < x \land 1 \neq 0]) \equiv x = 0 \lor x = 1 \{\text{quantifier elimination}\}$$

$$\varphi_2(x) \equiv \ldots$$

$$\equiv x \geq 0 \land x \leq 2$$

Now we prove that these predicates characterize the $A_i$ sets. Without imposing special conditions on the CRSs, we can only prove that the $\varphi_i$ predicates offer sufficient conditions for belonging to the $A_i$ sets. More formally,
\{ \bar{x} \in \mathbb{N}^s \mid \varphi_i(\bar{x}) \} \subseteq A_i \text{ for each } i \in \mathbb{N}. \text{ Strict inclusion may hold, in particular, when there are elements in the domain such that the evaluation of } T_h \text{ gets stuck, as the following example shows.}

**Example 7.** Given the following CRS:

\[
\begin{align*}
T_h(x) &= 0 \quad \{ x = 0 \} \\
T_h(x) &= 1 + T_h(x') \quad \{ x \geq 2 \land (x' = 0 \lor x' = 1) \}
\end{align*}
\]

We get \( T_h = [0 \mapsto \{0\}, 1 \mapsto \emptyset] \cup [i \mapsto \{1\} \mid i \geq 2] \) and hence, \( A_0 = \{0\} \) and \( A_i = \{0, 2, 3, 4, \ldots\} \) for each \( i \geq 1 \). However, by applying the corresponding definition, we obtain \( \varphi_i \equiv x = 0 \) for each \( i \in \mathbb{N} \). This is an exact approximation to \( A_i \) only when \( i = 0 \).

We can ensure that the \( \varphi \) predicates actually characterise the \( A_i \) by imposing some mild conditions on our CRSs, namely, that every vector \( \bar{x} \) satisfies at least one of the guards in the CRS.

**Theorem 1.** Given the CRS in (3), assume that \( \psi_h(\bar{x}) \lor \exists \bar{x}' . \psi_r(\bar{x}, \bar{x}') \) holds for every \( \bar{x} \in \mathbb{N}^s \). If \( T_h : \mathbb{N}^s \to \mathcal{P}(\mathbb{N}_\infty) \) is the relation defined by this CRS, the following holds for each \( i \in \mathbb{N} \):

\[
A_i = \{ \bar{x} \in \mathbb{N}^s \mid \varphi_i(\bar{x}) \}
\]

**Proof.** By induction on \( i \). The condition \( \psi_h(\bar{x}) \lor \exists \bar{x}' . \psi_r(\bar{x}, \bar{x}') \) is only needed for proving the \( \subseteq \) inclusion. The \( \supseteq \) inclusion holds without special provisions. \( \square \)

## 5 Searching for Test Nodes: Gradient-based Method

Recall that \( s \) is the dimension of the problem, that is the amount of variables on which \( h \) depends. The sets \( A_i \) correspond to predicates \( \varphi_i(\bar{x}) \) over variables \( \bar{x} = (x_1, \ldots, x_s) \). The graph of \( h(\bar{x}) \) in \( s+1 \)-dimensional space is presented via the collection of sets \( A'_i := (A_i, i) = \{(\bar{x}, i) \mid \bar{x} \in A_i\} \), which, informally speaking, form upside-down terraces. This surface is explained by the fact that \( A_i \subseteq A_{i+1} \). An upper bound on \( h \) “covers” the graph of \( h \). Intuitively, the monotonicity behavior of a good upper bound and the monotonicity behavior of \( h \) coincide, in the sense that the gradient of the bound at a point on the edge \( A_i \) is the almost the same as the “gradient” of \( h \) at this point. The gradient of a smooth scalar function \( f(\bar{x}) \) at a point \( \bar{x} \) shows the direction of the greatest rate of increase of the function. It is defined as the vector of the derivatives: \( \nabla(f) := \left( \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_s} \right) \). The graph of \( h \) is not smooth, therefore here the notion of the gradient at the point \( \bar{x} \in A_i \) is intuitive and taken as direction to the closest point on the next-level terrace \( B_{i+1} := A_{i+1} \setminus A_i \). The gradient-based method of finding test nodes mimics climbing up from \( A'_i \) to \( A'_{i+1} \). Based on this intuition, we propose the following procedure of finding test nodes.

1. **Inputs:**

105
the amount of levels $l$, and sets $A_i$, $0 \leq i \leq l$.
- the degree $d$ of a polynomial upper bound.
- the amount of climbing routes $r$ such that $r \times l \geq N(s, d)$, where $N(s, d)$ is the amount of coefficients of the polynomial in $s$ variables of the degree $d$.
- the initial terrace $A_{i_0}$. In the simplest case $i_0 = 0$.
- some $r$ points $\bar{x}_{i_0, j}$ on the initial terrace, where $j$ is the route counter. If the set $A_{i_0}$ is linear then its vertices may be chosen as initial points.

2. Take $i := i_0 + 1$, where $i$ is the level counter.
3. Take $j := 1$.
4. The next point $\bar{x}_{ij}$ on the $j$-th route is computed as the closest to $\bar{x}_{i-1,j}$ point on $B_i := A_i \setminus A_{i-1}$:

$$\bar{x}_{ij} := \text{argmin}_y \rho(\bar{x}_{i-1,j}, \bar{y})$$
$$\bar{y} \in B_i.$$

5. Repeat the procedure for all $j = 1, \ldots, r$.
6. Repeat the procedure for all $i = i_0 + 1, \ldots, l$.
7. Choose from the obtained nodes a subset that satisfies NCA condition.
8. Solve the corresponding linear system for the coefficients of a candidate $T^+_h(\bar{x})$.
9. Output: The candidate $T^+_h(\bar{x})$.

Now, let’s see how this procedure works for the recurrence for our example 1. Calculations have been implemented in a script for computer algebra reduce [12]. In principle, given a CRS, similar reduce-script may be generated automatically.

1. Inputs:
   - $l := 7$ and the sets $A_i$ are generated by their definition using quantifier-elimination procedure implemented in the reduce-package redlog [10]:

   \begin{align*}
   A_0 &: |x = 0 \land y = 0 \\
   A_1 &: |x = 0 \land y = 0 \lor x - 1 = 0 \land y = 0 \lor x = 0 \land y - 1 = 0 \\
   A_2 &: |A_1 \lor x - 1 = 0 \land y - 1 = 0 \lor x = 0 \land y - 2 = 0 \\
   A_3 &: |A_2 \lor x - 1 = 0 \land y - 2 = 0 \lor x - 2 = 0 \land y = 0 \lor x = 0 \land y - 3 = 0 \\
   A_4 &: |A_3 \lor x - 1 = 0 \land y - 3 = 0 \lor x - 2 = 0 \land y - 1 = 0 \lor x = 0 \land y - 4 = 0 \\
   A_5 &: |A_4 \lor x - 1 = 0 \land y - 4 = 0 \lor x - 2 = 0 \land y - 2 = 0 \lor x = 0 \land y - 5 = 0 \\
   A_6 &: |A_5 \lor x - 1 = 0 \land y - 5 = 0 \lor x - 2 = 0 \land y - 3 = 0 \lor x - 3 = 0 \land y = 0 \\
   A_7 &: \begin{cases}
   x = 0 \land y - 6 = 0 \\
   \lor x = 0 \land y - 7 = 0
   \end{cases}
   \end{align*}

   - $d := 2$,
   - $r := 3$,
   - the initial level is $i_0 := 3$.
   - the initial points are the vertices of $A_3$: $(1, 2), (2, 0), (0, 3)$.

2. The corresponding generated routes are (with levels from 4 to 7 incl.):

   1–st: $(1, 3), (1, 4), (1, 5), (1, 6)$
   2–nd: $(2, 1), (2, 2), (2, 3), (2, 4)$
   3–rd: $(0, 4), (0, 5), (0, 6), (0, 7)$
3. From these routes we pick up the following nodes: (1, 3) on level 4, (2, 1) on level 4 and (2, 2) on level 5, and (0, 4), (0, 5), (0, 6) on levels 4, 5, 6 respectively.

4. The corresponding linear system is given by the interpolation conditions

\[ T^+_h(1,3) = 4, \quad T^+_h(2,1) = 4, \quad T^+_h(2,2) = 5, \quad T^+_h(0,4) = 4, \quad T^+_h(0,5) = 5, \quad T^+_h(0,6) = 6. \]

E.g., the first condition gives the first equation

\[ a_{20} + 9a_{02} + 3a_{11} + a_{10} + 3a_{01} + a_{00} = 4. \]

The solution of the system is \( a_{20} = a_{10} = \frac{1}{2}, \quad a_{01} = 1 \) and the remaining coefficients are all zeros.

5. Output: The candidate \( T^+_h(\bar{x}) = \frac{x^2}{2} + \frac{x}{2} + y. \) It passes the checking shown in Sec. 6.

The obtained routes and the graph of \( T^+_h(x, y) \) are given in Fig. 1.

![Fig. 1. The obtained 3 routes and the graph of \( T^+_h(\bar{x}) = \frac{x^2}{2} + \frac{x}{2} + y \) that in this case coincides with \( h(x, y). \)](image-url)

6 Proving the Bound Correct

A sufficient condition under which the upper bound \( T^+_h(\bar{x}) \) obtained by interpolation is an upper bound to the height \( h(\bar{x}) \) of the evaluation trees is given by the conjunction of:

\[
\forall \bar{x}. T^+_h(\bar{x}) \geq 0
\]

\[
\forall \bar{x}, \bar{x}'. \psi_r(\bar{x}, \bar{x}') \Rightarrow (T^+_h(\bar{x}) \geq 1 + T^+_h(\bar{x}'))
\]

107
Assuming that $\forall \bar{x}. h(\bar{x}) \neq \infty$ (otherwise, there would be no bound for $h(\bar{x})$), the proof is a straightforward induction on $h(\bar{x})$.

But by considering the variables $\bar{x}, \bar{x}'$ as real numbers, by knowing that $\psi_r(\bar{x}, \bar{x}')$ is a conjunction of linear inequalities, and that $T_h^+$ is a multivariate polynomial of arbitrary degree, these formulas are decidable in Tarski’s theory of real closed fields [24]. There are a number of tools available implementing improved versions of Tarski’s procedure. For instance, QEPCAD [6] is free and offers an up-to-date version of Collins’ algorithm [8]. If the formulas hold for real numbers, they will also hold for natural ones.

Should this check fail, this could indicate that the degree chosen for the inferred polynomial is not enough. If the degree $d$ is chosen very high at the beginning, this would result in needing many test points, and so in more computation time. But the polynomial inferred could be of a lower degree than $d$ because the coefficients of higher degree terms would be equal to 0. A possible strategy is to start with a low degree such as $d = 2$, and then increase $d$ by 1 or 2 at each iteration until either a degree succeeds or some time-out expires. In the latter case, we would report a fail to infer the bound.

7 Conclusions and Related Work

In this paper we have applied polynomial interpolation-based techniques in order to extend the PUBS recurrence solver, so that it can deal with a broader set of CRSs.

Related Work

We have taken the work described in [2] as our point of reference. In a more recent work [4] the authors improve the precision of PUBS by considering worst- and best-case bounds to the cost of each loop iteration. The ideas described in this paper are orthogonal to those in [4] and can also be applied there.

In a different direction, COSTA has improved its memory analysis in order to take different models of garbage collection into account [3]. However, the authors claim that this extension does not require any changes to the recurrence solver PUBS. Thus, the techniques presented here should also fit with this extension.

In the field of functional languages, a seminal paper on static inference of memory bounds is [15]. A special type inference algorithm generates a set of linear constraints which, if satisfiable, they build a safe linear bound on the heap consumption.

One of the authors extended this type system in [13] in order to infer polynomial bounds. Surprisingly, the constraints resulting from the new type system are still linear ones. Although not every polynomial can be inferred by this system, the work was a remarkable step forward in the area. The language used is still functional, first-order and eager, but the resource inferred is a parameter. It could be either memory or time depending on some constants attached to the typing rules. A limitation of this work is that the inferred polynomials, even if
they are multivariate ones, must not have multivariate terms. This limitation is removed in a more recent work [14].

The application of polynomial interpolation techniques makes it possible to derive polynomial complexity without any restriction in advance on the kind of polynomials. With interpolation polynomials can be multivariate and non-monotonic. For size analysis of functional languages several interpolation results have been developed in the AHA Project [11]. First, a size analysis type system is developed together with language constraints such that sized type checking can be shown to be decidable. With polynomial interpolation type inference is made possible [16]. The full sized type system is given in [20]. In [23] it is shown how the basic type system, which is defined for list structures only, can be extended to allow algebraic data types. The size analysis systems give precise size functions. It has been shown that also general polynomial lower and upper bounds can be derived using polynomial interpolation [21].

Polynomial interpolation has also been applied to non-functional languages. For Java an analysis was made to derive ranking functions for loops [22].

References

Ranking Functions for Loops with Disjunctive Exit-Conditions

Rody Kersten$^1$ and Marko van Eekelen$^{1,2}$

$^1$ Institute for Computing and Information Sciences (iCIS), Radboud University Nijmegen
$^2$ School for Computer Science, Open University of the Netherlands

\{R.Kersten,M.vanEekelen\}@cs.ru.nl

Abstract. Finding ranking functions for the loops in a program is a prerequisite for proving its termination and analysing its resource usage. From its ranking function one easily derives a symbolic upper bound on the number of iterations of a loop. Such symbolic loop bounds can be used to derive concrete time and memory-usage bounds for complete programs.

This paper builds upon an earlier paper in which a polynomial interpolation based ranking function inference method is introduced for loops with exit conditions that are expressions in propositional logic over arithmetical (in)equalities.

We show that this earlier method is not applicable for certain loops: loops in which so-called condition jumping can occur. We define condition jumping and give an algorithm to detect it using symbolic execution and an SMT solver. We show how the earlier method can be adapted to be applicable also in the presence of condition jumping. As a result polynomial interpolation can be applied on a larger class of programs to infer polynomial ranking functions.

Keywords: Loop Bound, Ranking Function, Resource Analysis, Symbolic Execution, SMT solver

1 Introduction

Where software is used in safety-critical applications, it is important to derive and prove certain properties of this software, such as timing constraints and resource-usage bounds. It may be vital to an application that it runs within a fixed amount of time (e.g. when deploying an airbag) or memory (e.g. in embedded systems with limited capacity).

In order to prove termination of a piece of software or, even harder, calculate bounds on runtime or usage of resources such as heap-space or energy, finding bounds on the number of iterations that the loops in the software can make is a first step. While in some cases a loop may iterate a fixed number of times, often its execution will depend on user input. Therefore we consider symbolic loop bounds, or ranking functions.
A ranking function is a function over (some of) the program variables used in the loop, that decreases at each iteration and is bounded by zero. Listing 1 shows a simple while loop. Although $100 - i$ is a perfectly fine ranking function as well, the most precise one for this loop is $15 - i$. This gives the exact number of iterations the loop will make, for arbitrary $i$.

```
1 while (i < 15) {
2     i++;
3 }
```

Listing 1. A simple while loop, with most precise ranking function $15 - i$.

This paper builds upon an earlier paper in which a polynomial interpolation based ranking function inference method is introduced for loops with exit conditions that are expressions in propositional logic over arithmetical (in)equalities.

We show that this earlier method is not applicable for certain loops: loops in which so-called condition jumping can occur. The basic method, which considers only loops where the exit conditions are conjunctions over numerical (in)equalities, is described in Sect. 2. This method is extended to disjunctive exit conditions in Sect. 3, which requires piecewise ranking functions. At any point in execution, one part of a disjunctive exit condition may hold, then after execution of the loop body, another part of that condition may hold. This complicates our solution of splitting up the disjunctive parts of the condition. A formal definition of this condition jumping is given in Sect. 4, along with a method to detect it and an extension to the basic method which enables it to infer ranking functions also in the presence of condition jumping. Future work is discussed in Sect. 5 and related work in Sect. 6. The paper is concluded in Sect. 7.

## 2 Test Based Inference of Polynomial Ranking Functions for Loops using Polynomial Interpolation

In this section we describe what we will henceforth refer to as the basic method, first presented in [18]. In the basic method, we consider only loops for which the exit conditions are conjunctions over arithmetical (in)equalities:

$$\bigwedge_{i=1}^{n_i} (e_{ti} b e_{ri})$$

with $b \in \{<, >, =, \neq, \leq, \geq\}$.

The method works in the following steps:

1. Instrument the loop with a counter
2. Run test on a well-chosen set of input values
3. Find the polynomial interpolation of the results
Here, well-chosen means that test-nodes have to be picked such that there exists a unique interpolating polynomial. This is the reason we can refer to the polynomial interpolation in step 3.

Polynomial interpolation theory is described in Sect. 2.1. In Sect. 2.2, the method is further clarified using an example. In Sect. 2.3, soundness is discussed.

2.1 Polynomial Interpolation

When the result of a polynomial function \( p \) is known for a certain set of test values \( W \), the values of its coefficients \( a_0, \ldots, a_d \) can be derived (where \( d \) is the degree of the polynomial). Such a polynomial, which interpolates the test results, exists and is unique under certain conditions on the test data. These conditions are explored in polynomial interpolation theory [7].

In case \( p \) is a polynomial over a single variable \( z \), this condition is well-known: \( W \) must contain \( d + 1 \) pairwise different test-nodes. More specifically, if:

\[
p(z) = a_0 + a_1 z + \ldots + a_d z^d
\]

And for a set of \( d + 1 \) pairwise different test-nodes \( z_0, \ldots, z_d \) we know the values of the polynomial function \( p(z_0), \ldots, p(z_d) \), then we can find the unknown coefficients \( a_0, \ldots, a_d \). These values of the polynomial function determine a system of linear equations with respect to the coefficients. This system of equations can be presented in matrix-form as follows:

\[
\begin{pmatrix}
1 & z_0 & \cdots & z_0^{d-1} & z_0^d \\
1 & z_1 & \cdots & z_1^{d-1} & z_1^d \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
1 & z_{d-1} & \cdots & z_{d-1}^{d-1} & z_{d-1}^d \\
1 & z_d & \cdots & z_d^{d-1} & z_d^d \\
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
\vdots \\
a_{d-1} \\
a_d \\
\end{pmatrix}
= 
\begin{pmatrix}
p(z_0) \\
p(z_1) \\
\vdots \\
p(z_{d-1}) \\
p(z_d) \\
\end{pmatrix}
\]

There exists a unique solution to this system of equations and thus a unique polynomial interpolation in case the Vandermonde determinant of the matrix is non-zero. This is the case for pairwise different \( z_0, \ldots, z_d \).

The condition under which the Vandermonde determinant is non-zero in the multivariate case is more complex. In [7], a series of node configurations under which a unique interpolation exists is presented. We use the simplest version: Node Configuration A, or NCA. It is only presented briefly here. It is described in more detail in [18], in which we first presented the basic method, and in [17], in which it is applied to test-data generation for size analysis of functional programs.

First, recall that a polynomial \( p(z_1, \ldots, z_k) \) of degree \( d \) and dimension \( k \) (the number of variables) has \( N_d^k = \binom{d+k}{k} \) coefficients. For the two-dimensional case, NCA is defined as follows:

\( N_d^2 \) nodes forming a set \( W \subset \mathbb{R}^2 \) lie in 2-dimensional NCA if there exist lines \( \gamma_1, \ldots, \gamma_{d+1} \) in the space \( \mathbb{R}^2 \), such that \( d + 1 \) nodes of \( W \) lie on \( \gamma_{d+1} \) and \( d \) nodes
of \( W \) lie on \( \gamma_d \setminus \gamma_{d+1}, \ldots, \) and finally 1 node of \( W \) lies on \( \gamma_1 \setminus (\gamma_2 \cup \ldots \cup \gamma_{d+1}) \).

A typical instance of such a configuration is a 2-dimensional grid. For dimensions \( k > 2 \) the NCA is defined inductively on \( k \):

\[
N_k^d \text{ nodes forming a set } W \subset \mathbb{R}^k \text{ lie in } k\text{-dimensional NCA if, for any } 0 \leq i \leq d, \text{ there is a } (k-1)\text{-dimensional hyperplane such that it contains } N_{d-i}^{k-1} \text{ of the given nodes lying in } (k-1)\text{-dimensional NCA for the degree } d-i \text{ and these nodes do not lie on any of the other hyperplanes.}
\]

Because we are applying the polynomial interpolation theory to the inference of ranking functions for loops, the test-nodes must also satisfy the exit condition of the analysed loop. An algorithm to find these test-nodes is described in [18]. It looks for test-nodes on a \( k \)-dimensional grid.

### 2.2 Quadratic Example

Consider the example in Listing 2. Its ranking function is the degree 2 polynomial \( a \cdot b - c + 1 \).

```java
while (a > 0 && c <= b && c > 0) {
    if (c == b) {
        a--; 
        c = 0;
    }
    c++;
}
```

**Listing 2.** A while loop with quadratic ranking function \( a \cdot b - c + 1 \).

The inference of a ranking function for the loop in Listing 2 is depicted in Fig. 1.

First, the loop is instrumented with a counter. The user inputs the expected degree 2 of the polynomial ranking function. Since there are 3 variables, a set of \( N_2^3 = 10 \) test-nodes in NCA is generated. By interpolating the results from test runs using these input values, the most precise quadratic ranking function \( a \cdot b - c + 1 \) is found.

### 2.3 Soundness

The presented method infers a hypothetical ranking function. It is not sound by itself, but requires an external verifier.

We have chosen Java as source language in our implementation, therefore we use JML to express the ranking functions. However, there is no reason why the procedure would not work in other languages, such as C. When using C, the C-equivalent of JML, ACSL [4] might be used, which contains the same construct that we use to express ranking functions in JML.
Inferred ranking functions are expressed in JML by defining a `decreases` clause on the loop. This is an expression which must decrease by at least 1 on each iteration and that remains greater than or equal to 0, see the JML reference manual [15]. It therefore forms an upper-bound on the number of iterations of the loop. An example is shown in Listing 3.

```java
//@ decreases i < 15 ? 15 - i : 0;
while (i < 15) {
    i++;
}
```

Listing 3. The loop from Listing 1, annotated with its ranking function
When the loop condition does not hold, the loop iterates zero times. Therefore the shown annotation actually expresses the maximum of $15 - i$ and 0.

In general, a ranking function $RF(\bar{v})$ for a loop with condition $b$ can be expressed as \textit{decreases} $b ? RF(\bar{v}) : 0$.

Such JML annotations can be verified by a variety of tools. In our experiments we have used the KeY tool [5]. The procedure described in this paper should be used in conjunction with such a prover to provide soundness.

### 3 Piecewise Ranking Functions

In this section we extend the set of considered loops to those with as exit condition \textit{any} propositional logical expression over arithmetical (in)equalitys, thus now including disjunctions. We give a slightly more formal and thorough definition of our solution as in [18]. We will see that for those loops for which the exit condition contains disjunctions, the ranking function will become \textit{piecewise}.

Note that in fact, any ranking function for a well-formed loop is a piecewise one, since there is always the piece where the exit condition does not hold and the loop iterates zero times. For instance, for the loop in Listing 1, the ranking function is actually:

\[
\begin{cases} 
15 - i & \text{if } (i < 15) \\
0 & \text{else}
\end{cases}
\]  

(1)

This is of course a trivial case. A more involved example of a loop for which a piecewise ranking function can be defined is shown in Listing 4.

```
1 while ( ((i>0) && i<20) || i>50) {
2    if (i>50) i--; 
3    else i++;
4 }
```

Listing 4. While loop with a piecewise ranking function.

It’s ranking function is the following:

\[
\begin{cases} 
20 - i & \text{if } (i > 0) \land (i < 20) \\
i - 50 & \text{if } i > 50 \\
0 & \text{else}
\end{cases}
\]  

(2)

In this section we will show how the basic method is extended to treat loops with simple disjunctions in their exit conditions.

#### 3.1 Extending the Basic Method

The exit condition of the loop in Listing 4 can easily be split up into three disjunctive parts: $i > 0 \land i < 20 \land \neg(i > 50)$, $i > 50 \land \neg(i > 0 \land i < 20)$ and $(i > 0 \land i < 20) \land i > 50$. The latter condition is not satisfiable by any $i$ and is therefore removed.
Simplifying the remaining conditions leads to the following result: \( \text{Pieces} = \{(i > 0 \land i < 20), (i > 50)\} \), where \( \text{Pieces} \) defines the pieces of the piecewise polynomial ranking function. We can now execute the basic method for both of these pieces separately. This procedure leads to the ranking function in Eq. 2, as one would expect.

We will now formally define a generic method for inferring ranking functions for loops with disjunctive exit conditions, which we call \textit{DNF-splitting}.

The first step is to transform the exit condition into disjunctive normal form (DNF), using the laws of distribution and DeMorgan’s theorems. It thereafter thus has the form:

\[
\bigvee_{i=1}^{n} \left( \bigwedge_{j=1}^{m_i} (e_{ij} \ b_{e_{rij}}) \right)
\]

with \( b \in \{<, >, =, \neq, \leq, \geq\} \).

Let us shorten this to the following for readability:

\[
\bigvee_{i=1}^{n} b_i
\]

So, each \( b_i \) represents a logical conjunction over numerical (in)equalities. We can now split up the exit condition by applying the function \( \text{DNFsplit} : C_d \rightarrow \{C_{nd}\} \), where \( C_d \) is the type representing conditions of the form described above and \( \{C_{nd}\} \) is a collection of conditions in the form described in Sect. 2 (conjunctions over arithmetical (in)equalities).

\[
\text{DNFsplit}(b_1 \lor \ldots \lor b_n) := \left\{ \bigwedge_{b_i \in BP} b_i \land \bigwedge_{b_j \in B_{rest}} \neg b_j \mid BP \in \mathcal{P}([b_1, \ldots, b_n]) \setminus \emptyset \land B_{rest} = \{b_1, \ldots, b_n\} \setminus BP \right\}
\]

This transforms the condition \( b_1 \lor \ldots \lor b_n \) into a set \( \text{Pieces} \) of \( 2^n - 1 \) conjunctive conditions of type \( C_{nd} \). This set may contain unsatisfiable conditions. The unsatisfiability can be detected using an SMT solver and such conditions may safely be removed.

These conditions define the pieces of the piecewise polynomial ranking function. We can now apply the basic method separately for each of these pieces. If \( RF_p \) is the polynomial ranking function inferred for a piece \( p \in \text{Pieces} \), then this yields the following piecewise ranking function:

\[
\begin{cases}
RF_{p_1} & \text{if } p_1 \\
\ldots & \text{if } \ldots \\
RF_{p_m} & \text{if } p_m \\
0 & \text{else}
\end{cases}
\]  

(3)

In this piecewise polynomial ranking function, \( m \leq 2^n - 1 \), because unsatisfiable pieces have been removed.
### 3.2 Expressing Piecewise Ranking Functions in JML

By using the \( b ? e_1 : e_2 \) notation for conditionals, we are able to express a piecewise polynomial \textit{decreases} clause in JML. The piecewise ranking function 2 for the loop in Listing 4 is shown in Listing 5.

```jml
//@ decreases (i>0 && i<20) ? 20−i : ( i>50 ? i−50 : 0 );
while ((i>0 && i<20) || i>50) {
    if (i>50) i--;        
    else i++;
}
```

**Listing 5.** Piecewise ranking function 2 expressed as a JML annotation.

In general, a ranking function of the form in Eq. 3 can be expressed as
\[
decreases p_1 \mathbin{\text{RF}} p_1 : ( \ldots : (p_m \mathbin{\text{RF}} p_m : 0) ).
\]

### 4 Condition Jumping

Building on the definitions from the previous section, we here define a complication that arises, which we call \textit{condition jumping}. We also show how to detect its occurance and how to infer ranking functions even in the presence of condition jumping, which is our main contribution.

```jml
while ((i>0 && i<20) || i>22) {
    if (i>22) i--;        
    else i+=4;
}
```

**Listing 6.** While loop with jumping between the disjunctive conditions.

Consider the loop in Listing 6. Naively, one could say that its ranking function is the following (the notation \([n]\) means \(n\) “ceiled” or rounded up):

\[
\begin{align*}
\lfloor(20 − i)/4\rfloor & \text{ if } (i > 0) \land (i < 20) \\
n \mod 4 = 3 & \text{ else }
\end{align*}
\]

But, what if \(i\) is 19, 15, or any \(n \in [1, 19]\) with \(n \mod 4 = 3\)? Indeed, then there is an overflow from the first condition \((0 < i < 20)\) to the second one \((i > 22)\). We call this \textit{condition jumping}. Jumping from the second condition into the first one is not possible in this case.

Because of the presence of condition jumping, regular DNF-splitting does not suffice here. The set of nodes from which condition jumping occurs must be considered as a separate piece, as follows:

\[
\begin{align*}
\lfloor(20 − i)/4\rfloor + 1 & \text{ if } (i > 0) \land (i < 20) \land i \mod 4 = 3 \\
\lfloor(20 − i)/4\rfloor & \text{ if } (i > 0) \land (i < 20) \land i \mod 4 \neq 3 \\
n \mod 4 = 3 & \text{ else }
\end{align*}
\]

118
Figure 2 depicts condition jumping for a loop with exit condition \( b_1 \lor b_2 \). The total set of nodes satisfying \( b_1 \) is \( D_1 \). The total set of nodes satisfying \( b_2 \) is \( D_2 \). \( D_{1,2} \) is the subset of \( D_1 \) for which jumping to \( b_2 \) occurs, \( D_{1,1} \) is the subset for which jumping does not occur.

In the example, \( D_1 = [1, 19] \), \( D_2 = [23, \infty) \), \( D_{1,2} = \{n | n \in [1, 19] \land n \mod 4 = 3 \} \) and \( D_{1,1} = D_1 \setminus D_{1,2} \).

A method to detect condition jumping is described in Section 4.1. This method is then extended to detect all the nodes in \( D_{1,2} \) in Section 4.2, in order to infer a correct piecewise ranking function.

4.1 Detection of Condition Jumping using Symbolic Execution and SMT Solvers

To detect condition jumping in the example in Listing 6, we will first use symbolic execution [14] to capture the relation between the values of the program variables pre and post execution of the loop body. We can then use this relation as input to an SMT solver and check if one part of the condition is true pre-execution of the body and another part is true post-execution.

We will name this pre/post execution relation for a variable \( v \) the \( \text{next}_v \) function. The function \( \text{next}_i :: \text{Int} \rightarrow \text{Int} \) for the loop in from Listing 6 can be determined by symbolically executing the loop with value \( \alpha_i \) for \( i \). This results in the following symbolic post-execution value, which we will name \( \phi_i \):

\[
\phi_i(\alpha_i) = \begin{cases} 
\alpha_i - 1 & \text{if } \alpha_i > 22 \\
\alpha_i + 4 & \text{if } \neg(\alpha_i > 22)
\end{cases}
\]  

(6)

By replacing the \( \alpha \) symbol by \( i \), this easily translates to the \( \text{next}_i \) function we were looking for:

\[
\text{next}_i(i) = \begin{cases} 
i - 1 & \text{if } i > 22 \\
i + 4 & \text{if } \neg(i > 22)
\end{cases}
\]  

(7)

An SMT-LIB script to detect jumping in the example from Listing 6 is given in Listing 7. The function \( \text{next}_i :: \text{Int} \rightarrow \text{Int} \) from Equation 7 is defined on line 3. Then on line 4 we define the condition expressing that jumping occurs for this example and on line 5 we check satisfiability of this condition.
Let us now consider the general case. Condition jumping will be detected pairwise for conditions with multiple disjunctions. Here we thus consider a single pair of conditions, i.e. a while loop with exit condition \( b_1 \lor b_2 \). Here \( b_1 \) and \( b_2 \) are Boolean expressions ranging over \( CV \subseteq LV \subseteq PV \), where \( CV \) are the program variables in the condition, \( LV \) are the program variables in the loop and \( PV \) are all program variables.

For each \( v_i \in LV \), we can define an associated function \( next_{v_i} : T_{v_1} \to \ldots \to T_{v_i} \to \ldots \to T_{v_n} \to T_{v_i} \), where \( T_{v_i} \) is the type of \( v_i \) and \( n = |LV| \), which takes the values of all \( v \in LV \) as the state and computes the value of \( v \) after a single execution of the loop body in that state.

Such a function can be derived by symbolic execution of the loop body. Start by giving the variables \( v_1 \ldots v_n \) symbolic values \( \alpha_1 \ldots \alpha_n \). After the symbolic execution of the loop body, each variable \( v_i \) will now have a value which is a set of polynomials over the symbols \( \alpha_1 \ldots \alpha_n \) and constants, with associated path conditions, which capture branching. Effectively, this is again a piecewise polynomial. The function \( next_{v_i} \) is now obtained by replacing the \( \alpha \)'s by the corresponding program variables in this piecewise polynomial.

Once these functions have been derived, the question whether jumping from \( b_1 \) to \( b_2 \) is possible can be answered by any SMT-LIB conforming SMT-solver\(^3\) by determining the satisfiability of \( b_1(v_1, \ldots, v_n) \land b_2(next_{v_1}(LV), \ldots, next_{v_n}(LV)) \).

### 4.2 Generating Ranking Functions in the Presence of Condition Jumping

The SMT-LIB script in Listing 7 can be used to find a model for which jumping occurs by adding the expression \((\text{get-value } (i))\) between lines 5 and 6. A model is an instantiation of the variables for which the formula for which satisfiability is checked holds. In the SMT-LIB script from Listing 7, a model for \( i \) is 19.

Successively, the script in Listing 8 can now be used if there is any models other than \( i = 19 \) for which jumping occurs. The answer of the SMT solver is that the combination of propositions in this script is unsatisfiable. Thus, \( i = 19 \) is the only possible model.

\(^3\) For instance Z3, which can be used online at [http://research.microsoft.com/en-us/um/redmond/projects/z3/](http://research.microsoft.com/en-us/um/redmond/projects/z3/)
We can now see if there are any models from which the state $i = 19$ can be reached in a single iteration. The script in Listing 9 finds the model $i = 15$.

Subsequently and similarly, we can search for other nodes that can reach the state $i = 19$ in a single step, or that can reach the state $i = 15$. By repeating these steps, we can find the set $D_{1,2} = \{3, 7, 11, 15, 19\}$. These are the models from which jumping can occur.

The term model is very similar to the term node. Both refer to an instantiation of program variables with a specific set of values. The term model is common in the area of SMT-solvers, while the term node is commonly used for test-points in the area of interpolation. The relation is as follows: $\text{nodes} \subseteq \text{models}$, because a set of test-nodes that is deemed suitable for interpolation is chosen from the total set of models satisfying a loop its exit condition. We will refer to a model with the vector $\vec{v}$.

In general, the method described in Section 4.1 can be extended to detect all models from which condition jumping can occur, by first finding all models that can jump directly from $b_1$ to $b_2$ and then recursively finding models that can reach a model from this first set. This can be done by implementing the following algorithm around an SMT-solver. In this algorithm, $J$ is the set of models of which it is known that condition jumping occurs and $Q$ is a queue of models. We assume a function $\text{next} : M \to M$ (where $M$ is the type of a model), which applies to each variable $v_i$ in a model $\vec{v}$ its corresponding $\text{next}_{v_i}$ function.

Listing 8. SMT-LIB script to detect other models than $i = 19$ for which jumping occurs in the code of Listing 6.

Listing 9. SMT-LIB script to detect models that can reach the state where $i = 19$.
1. Is there a model $\bar{v}$ for which $b_1(\bar{v}) \land b_2(\text{next}(\bar{v})) \land \bar{v} \not\in J$?
   - SAT $\rightarrow$ Add $\bar{v}$ to $J$ and $Q$, goto 1.
   - UNSAT $\rightarrow$ Goto 2.

2. Q empty?
   - Yes $\rightarrow$ Done.
   - No $\rightarrow$ Goto 3.

3. Pop a model $\bar{q}$ off the queue $Q$. Is there a model $m$ for which $b_1(\bar{v}) \land \text{next}(\bar{v}) = \bar{q} \land \bar{v} \not\in J$?
   - SAT $\rightarrow$ Add $\bar{v}$ to $J$ and $Q$, goto 3.
   - UNSAT $\rightarrow$ Goto 2.

After execution, $J$ contains exactly all elements from $D_{1,2}$. Since here a queue is used, this algorithm implements a breadth-first search. This can easily be adapted to a depth-first search by using a stack.

Now that we know $D_{1,2}$, we can split the condition $b_1$ into two: $b_1(\bar{v}) \land \bar{v} \in D_{1,2}$ and $b_1(\bar{v}) \land \bar{v} \not\in D_{1,2}$. We can then apply the basic method to each of these disjunctive pieces.

### 4.3 Multi-Jumping

The algorithm described in the previous section detects jumping between two disjunctive parts of a condition. We need an umbrella algorithm to iteratively apply that pairwise algorithm in order to detect all jumping between all pieces.

For a condition $b_1, \ldots, b_n$, with respective sets $D_1, \ldots, D_n$ of all the models satisfying them, the following algorithm can be used:

1. DNF-split into $n$ conditions
2. For each $i$ and $j$, $1 \leq i < j \leq n$, detect jumping from $D_i$ to $D_j$. Build a list $J$ of jumping pairs $(D_x, D_y)$ for which condition jumping from $D_x$ to $D_y$ can occur.
3. If there are no more jumping pairs $(D_x, D_y)$ for which $D_x$ is unflagged, done! Else, goto 4.
4. Pop a jumping pair $(D_x, D_y)$ off $J$, for which $D_x$ is unflagged.
5. Apply the algorithm from Sect. 4.2 to find the set $D_{x,2}$ of all nodes in $D_x$ from which jumping to $D_y$ occurs and, dually, the set $D_{x,1}$ for which no jumping to $D_y$ occurs. Replace any condition pair $(D_x, D_z)$ in $J$ by $(D_{x,1}, D_{z,2})$. Add $(D_{x,2}, D_y)$ to $J$.
   - If $D_{x,1} = \emptyset$, flag $D_{x,2}$ as complete, goto 3.
   - Else, for any jumping pair $(D_z, D_x)$ in $J$ (i.e. for which jumping from $D_z$ to $D_x$ can occur), unflag $D_z$, detect jumping into $D_{x,1}$ and $D_{x,2}$ and update $J$ accordingly. Goto 3.
5 Future Work

5.1 Implementation and Case Study

We have implemented the basic method and DNF-splitting for loops without condition jumping in a tool called ResAna. This tool can be downloaded at http://www.resourceanalysis.cs.ru.nl. We intend to implement the condition jumping detection method described in this paper as well, and adapt the implementation so that it can generate ranking functions in the presence of condition jumping.

When this implementation is done, the procedure can also be tested on a case study, to see which loops it can now analyse that were too complex before.

5.2 Use SMT Solver in Node-Search

In this paper, we use an SMT solver to detect condition jumping and to remove unsatisfiable conditions from the pieces list generated by DNF-splitting. The currently used algorithm to find test-nodes is described in [18], which searches for nodes in a quite extensive way. Node-search might be optimised by expressing NCA in SMT-LIB syntax and incrementally asking an SMT-solver to find a node, similar to the algorithm presented in Sect. 4.2.

6 Related Work

Various other research results on bounding the number of loop iterations are described in the literature. However, most approaches generate concrete (numerical) bounds, as opposed to symbolic bounds. The methods that are able to infer symbolic loop bounds are limited to either bounds that depend linearly on program variables (the procedure described in this paper infers polynomial bounds) or that are constructed from monotonic subformulae. Several syntactical methods are discussed, that will be more efficient for simple cases, but less general. The procedure described in this paper can be seen as complementary for those methods. In case a syntactical method is not applicable to a certain loop, the more general method described here can be used.

Another common difference is that other approaches rely on handmade soundness proofs of their method, while we rely on a verification tool to ensure that the derived LBFs are correct.

In [10], Fulara et al use pattern-matching on abstract syntax trees (ASTs) to select one of several syntax-based schemes for generating decreases-clauses. If the AST matches a given pattern, then parameters from this pattern can be used to form a decreases-clause. The authors claim to cover 71% of all for-loops in a set of case studies.

Abstract interpretation, program slicing and invariant analysis are used by Ermedahl et al in [9] to infer numerical bounds for C programs. The bounds meant here are integers representing the number of times a certain block of code
is executed. The method can infer bounds for over 50% of the loops in a set of benchmarks.

A similar approach is taken by Lokuciejewski et al in [16], who combine abstract interpretation with polytope models to calculate numerical loop bounds for C programs. Both upper and lower bounds are calculated and the analysis is accelerated by using program slicing. Even though there are restrictive constraints on the loops that can be analysed, the authors claim that they can handle 99% of all for-loops in a set of benchmarks. Soundness or verification of the bounds are not discussed.

Abstract interpretation is also used in [8], in combination with flow analysis. Numerical bounds can be found for 84% of the loops in a benchmark suite. The method works on C programs.

In [6], Ben-Amram describes a method to derive global ranking functions, based on Size-Change Termination. Such a ranking function is required to decrease in each basic block of the program. He uses an abstraction called Monotonicity Constraints and represents them as graphs. Various algorithms are described that can be applied to these graphs to judge termination and construct ranking functions.

Gulwani uses “off-the-shelf linear invariant generation tools” to compute symbolic loop bounds in [11]. The authors experiment with different counter instrumentation methods and a technique they named “control-flow refinement”. Ranking functions are presented as inequations in loop invariants. Inference of invariants is based on linear arithmetic, but some limited use of non-linear terms is possible as well. Given a particular program, the base arithmetic may be extended by a finite set of non-linear operators together with reasoning rules for them. The inference system, first, introduces a fresh variable for each non-linear operator, then deals with linear combinations of such variables (and usual arithmetic variables). The operators and the rules are chosen e.g. by a user, who knows which sort of invariants one can expect in the given code.

In [12], Gulwani and Zuleger present another method to derive ranking functions. First, a transition system representing the program is built. Then, patterns are matched against this transition system. This leads to disjunctive ranking functions for all transitions, which can successively be combined into a global ranking function. This method reputedly achieves better results for nested loops and is able to compute ranking functions for 76% of the loops in a .Net base-class library. They use an SMT solver to verify ranking functions over loop-free program fragments.

Hunt et al discuss the expression of manually conceived ranking functions in JML, their verification using KeY and the combination with data-flow analysis in [13]. This article is an important motivation for our work. What is “missing” in the method is the automated inference of ranking functions, which we supply.

In [1], Albert et al describe a system of generating and solving cost recurrence relations. These relations define functions that represent upper bounds on time or memory usage by a program. To solve a recurrence relation means to find a closed, i.e. a recursion-free, form of the corresponding function. Terms in
the system represent \textit{monotonic} real functions and, besides monotonically increasing polynomials, contain the exponent and the logarithmic functions. Their method is implemented in the COSTA system, which is described in [2]. In [3], an approach that is similar to ours in ResAna [18] is taken, in the combination of COSTA with the KeY tool. The results that COSTA gives are output as JML annotations, that may then be verified using KeY.

7 Conclusions

We have shown that the previously published polynomial interpolation based ranking function inference method is not applicable for certain loops: loops in which so-called condition jumping can occur. We have given a definition of condition jumping and presented an algorithm to detect it using symbolic execution and an SMT solver. Also, the earlier method has been adapted to be applicable also in the presence of condition jumping. This extension makes the method more powerful, in the sense that it can now infer symbolic loop bounds also for loops in which this condition jumping occurs.

Ranking functions for loops can be combined with ranking functions for the other statements and constructions in a program to create a global ranking function. The existence of a global ranking function proves termination of the program.

Also, ranking functions for loops are vital to an analysis of resource-usage. For instance, if a loop with ranking function \( RF(\vec{v}) \) contains the statement \texttt{new ObjOnHeap}, then we know that the execution of that loop will consume \( RF(\vec{v}) \cdot \text{size}(\text{ObjOnHeap}) \) bytes of heap-space. Other examples of resources that are often consumed proportionally to a symbolic loop bound include time, network traffic and energy.

It is therefore crucial for resource and termination analysis that (precise) ranking functions for all loops in a program can be defined, not just the simple linear cases or those that match a user-defined pattern. The basic polynomial interpolation method already formed a more general solution and with the extension presented in this paper, we are one step further in achieving this goal.

References