Liberal Typing for Functional Logic Programs*

(Extended version)
Technical Report SIC-06-10, 2010

Francisco López-Fraguas, Enrique Martin-Martin, and Juan Rodríguez-Hortalá

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, emartinm@fdi.ucm.es, juanrh@fdi.ucm.es

Abstract. We propose a new type system for functional logic programming which is more liberal than the classical Damas-Milner usually adopted, but it is also restrictive enough to ensure type soundness. Starting from Damas-Milner typing of expressions we propose a new notion of well-typed program that adds support for type-indexed functions, existential types, opaque higher-order patterns and generic functions—as shown by an extensive collection of examples that illustrate the possibilities of our proposal. In the negative side, the types of functions must be declared, and therefore types are checked but not inferred. Another consequence is that parametricity is lost, although the impact of this flaw is limited as "free theorems" were already compromised in functional logic programming because of non-determinism.

Keywords: Type systems, functional logic programming, generic functions, type-indexed functions, existential types, higher-order patterns.

1 Introduction

Functional logic programming. Functional logic languages [9] like TOY [18] or Curry [10] have a strong resemblance to lazy functional languages like Haskell [13]. A remarkable difference is that functional logic programs (FLP) can be non-confluent, giving raise to so-called *non-deterministic functions*, for which a *call-time choice* semantics [6] is adopted. The following program is a simple example, using natural numbers given by the constructors z and s—we follow syntactic conventions of some functional logic languages where function and constructor names are lowercased, and variables are uppercased—and assuming a natural definition for *add*: $\{\ f\ X \to X,\ f\ X \to s\ X,\ double\ X \to add\ X\ X\ \}$. Here, f is non-deterministic ($f\ z$ evaluates both to z and $s\ z$) and, according to call-time choice, *double (f z)* evaluates to z and $s\ (s\ z)$ but not to $s\ z$. Operationally, call-time choice means that all copies of a non-deterministic subexpression ($f\ z$ in the example) created during reduction share the same value.

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, S2009TIC-1465 and UCM-BSCH-GR58/08-910502.

In the HO-CRWL[1] approach to FLP [7], followed by the TOY system, programs can use *HO-patterns* (essentially, partial applications of symbols to other patterns) in left hand sides of function definitions. This corresponds to an *intensional* view of functions, i.e., different descriptions of the same 'extensional' function can be distinguished by the semantics. This is not an exoticism: it is known [17] that extensionality is not a valid principle within the combination of HO, non-determinism and call-time choice. It is also known that *HO-patterns* cause some bad interferences with types: [8] and [16] considered that problem, and this paper improves on those results.

All those aspects of FLP play a role in the paper, and Sect. 3 uses a formal setting according to that. However, most of the paper can be read from a functional programming perspective leaving aside the specificities of FLP.

Types, FLP and genericity. FLP languages are typed languages adopting classical Damas-Milner types [5]. However, their treatment of types is very simple, far away from the impressive set of possibilities offered by functional languages like Haskell: type and constructor classes, existential types, GADTs, generic programming, arbitrary-rank polymorphism ... Some exceptions to this fact are some preliminary proposals for type classes in FLP [22,19], where in particular a technical treatment of the type system is absent.

The term *generic programming* is not used in the literature in a unique sense. With it, we refer generically to any situation in which a program piece serves for a family of types instead of a single concrete type.

Parametric polymorphism as provided by Damas-Milner system is probably the main contribution to genericity in the functional programming setting. However, in a sense it is 'too generic' and leaves out many functions which are generic by nature, like equality. Type classes [25] were invented to deal with those situations. Some further developments of the idea of generic programming [11] are based on type classes, while others [12] have preferred to use simpler extensions of Damas-Milner system, such as GADTs [3,24]. We propose a modification of Damas-Milner type system that accepts natural definitions of intrinsically generic functions like equality. The following example illustrates the main points of our approach.

An introductory example. Consider a program that manipulates Peano natural numbers, booleans and polymorphic lists. Programming a function *size* to compute the number of constructor occurrences in its argument is an easy task in a type-free language with functional syntax:

size true $\to$ *s z* *size false* $\to$ *s z*
size z $\to$ *s z* *size (s X)* $\to$ *s (size X)*
size nil $\to$ *s z* *size (cons X Xs)* $\to$ *s (add (size X) (size Xs))*

However, as far as *bool*, *nat* and $[\alpha]$ are different types, this program would be rejected as ill-typed in a language using Damas-Milner system, since we obtain contradictory types for different rules of *size*. This is a typical case where one wants some support for genericity. Type classes certainly solve the problem

[1] CRWL [6] stands for *Constructor Based Rewriting Logic*; HO-CRWL is a higher order extension of it.

if you define a class *Sizeable* and declare *bool, nat* and $[\alpha]$ as instances of it. GADT-based solutions would add an explicit representation of types to the encoding of *size* converting it into a so-called *type-indexed* function [12]. This use of GADTs is also supported by our system (see the *show* function in Ex. 1 and *eq* in Fig 4-b later), but the interesting point is that our approach allows also a simpler solution: the program above becomes well-typed in our system simply by declaring *size* to have the type $\forall\alpha.\alpha \to nat$, of which each rule of *size* gives a more concrete instance. As a consequence of this relaxed type for $size$, the expression $size\ e$ is well-typed for any well-typed e, even if there is no applicable program rule and its evaluation leads to a pattern matching failure. Another issue is that types for functions must be explicitly supplied for checking that rules are well-typed. A detailed discussion of the advantages and disadvantages of our approach appears in Sect. 6 (see also Sect. 4).

The proposed well-typedness criterion requires only a quite simple additional check over usual type inference for expressions, but here 'simple' does not mean 'naive'. Imposing the type of each function rule to be an instance of the declared type is a too weak requirement, leading easily to type unsafety. As an example, consider the rule $f\ X \to not\ X$ with the assumptions $f : \forall\alpha.\alpha \to bool$, $not : bool \to bool$. The type of the rule is $bool \to bool$, which is an instance of the type declared for f. However, that rule does not preserve the type: the expression $f\ z$ is well-typed according to f's declared type, but reduces to the ill-typed expression $not\ z$. Our notion of well-typedness, roughly explained, requires also that right-hand sides of rules do not restrict the types of variables more than left-hand sides, a condition that is violated in the rule for f above. Def. 2 in Sect. 3.3 states that point with precision, and allows us to prove type soundness for our system.

Contributions. We give now a list of the main contributions of our work, presenting the structure of the paper at the same time:

• After some preliminaries, in Sect. 3 we present a novel notion of well-typed program for FLP that induces a simple and direct way of programming type-indexed and generic functions. The approach supports also existential types, opaque HO-patterns and GADTs.

• Sect. 4 is devoted to the properties of our type system. We prove that well-typed programs enjoy *type preservation*, an essential property for a type system; then by introducing *failure* rules to the formal operational calculus, we also are able to ensure the *progress* property of well-typed expressions. Based on those results we state type soundness.

• In Sect. 5 we give a significant collection of examples showing the interest of the proposal. These examples cover type-indexed functions, existential types, opaque higher-order patterns and generic functions. None of them is supported by existing FLP systems.

• Our well-typedness criterion goes far beyond the solutions given in previous works [8,16] to type-unsoundness problems of the use of *HO-patterns* in function definitions. We can type equality, solving known problems of *opaque decomposition* [8] (Sect. 5.1) and, most remarkably, we can type the *apply* function

appearing in the HO-to-FO translation used in standard FLP implementations (Sect. 5.2).

• Finally we discuss in Sect. 6 the strengths and weaknesses of our proposal, and we end up with some conclusions in Sect. 7.

2 Preliminaries

We assume a signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint sets of *data constructor* and *function* symbols resp., all of them with associated arity. We write CS^n (resp. FS^n) for the set of constructor (function) symbols of arity n, and if a symbol h is in CS^n or FS^n we write $ar(h) = n$. We consider a special constructor $fail \in CS^0$ to represent pattern matching failure in programs as it is proposed for GADTs [3,23]. We also assume a denumerable set $\mathcal{DV}$ of *data variables* X. Fig. 1 shows the syntax of *patterns* $\in Pat$—our notion of values—and *expressions* $\in Exp$. We split the set of patterns in two: *first order patterns* $FOPat \ni fot ::= X \mid c\ fot_1 \dots fot_n$ where $ar(c) = n$, and *higher order patterns* $HOPat = Pat \setminus FOPat$, i.e., patterns containing some partial application of a symbol of the signature. Expressions $c\ e_1 \dots e_n$ are called *junk* if $n > ar(c)$ and $c \neq fail$, and expressions $f\ e_1 \dots e_n$ are called *active* if $n \geq ar(f)$. The set of *free variables* of an expression—$fv(e)$—is defined in the usual way. Notice that since our let expressions do not support recursive definitions the binding of the variable only affect e_2: $fv(let\ X = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus \{X\})$. We say that an expression e is *ground* if $fv(e) = \emptyset$. A *one-hole context* is defined as $\mathcal{C} ::= [] \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C}$. A *data substitution* θ is a finite mapping from data variables to patterns: $[\overline{X_n/t_n}]$. Substitution application over data variables and expressions is defined in the usual way. The empty substitution is written as id. A *program rule* r is defined as $f\ \overline{t_n} \to e$ where the set of patterns $\overline{t_n}$ is linear (there is not repetition of variables), $ar(f) = n$ and $fv(e) \subseteq \bigcup_{i=1}^{n} var(t_i)$. Therefore, extra variables are not considered in this paper. The constructor *fail* is not supposed to occur in the rules, although it does not produce any technical problem. A program $\mathcal{P}$ is a set of program rules: $\{r_1, \dots, r_n\} (n \geq 0)$.

For the types we assume a denumerable set $\mathcal{TV}$ of *type variables* α and a countable alphabet $\mathcal{TC} = \bigcup_{n \in \mathbb{N}} \mathcal{TC}^n$ of *type constructors* C. As before, if $C \in \mathcal{TC}^n$ then we write $ar(C) = n$. Fig. 1 shows the syntax of *simple types* and *type-schemes*. The set of *free type variables (ftv)* of a simple type τ is $var(\tau)$, and for type-schemes $ftv(\forall \overline{\alpha_n}.\tau) = ftv(\tau) \setminus \{\overline{\alpha_n}\}$. We say a type-scheme σ is *closed* if $ftv(\sigma) = \emptyset$. A *set of assumptions* $\mathcal{A}$ is $\{\overline{s_n : \sigma_n}\}$, where $s_i \in CS \cup FS \cup \mathcal{DV}$. We require set of assumptions to be *coherent* wrt. CS, i.e., $\mathcal{A}(fail) = \forall \alpha.\alpha$ and for every c in $CS^n \setminus \{fail\}$, $\mathcal{A}(c) = \forall \overline{\alpha}.\tau_1 \to \dots \to \tau_n \to (C\ \tau'_1 \dots \tau'_m)$ for some type constructor C with $ar(C) = m$. Therefore the assumptions for constructors must correspond to their arity and, as in [3,23], the constructor *fail* can have any type. The union of sets of assumptions is denoted by $\oplus$: $\mathcal{A} \oplus \mathcal{A}'$ contains all the assumptions in $\mathcal{A}'$ and the assumptions in $\mathcal{A}$ over symbols not appearing in $\mathcal{A}'$. For sets of assumptions $ftv(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. Notice that type-schemes

Data variables		$X, Y, Z, \ldots$	Patterns	$t ::= X$
Type variables		$\alpha, \beta, \gamma, \ldots$		$\mid\ c\ t_1 \ldots t_n$ if $n \leq ar(c)$
Data constructors		c		$\mid\ f\ t_1 \ldots t_n$ if $n < ar(f)$
Type constructors		C	Simple Types	$\tau ::= \alpha$
Function symbols		f		$\mid\ C\ \tau_1 \ldots \tau_n$ if $ar(C) = n$
				$\mid\ \tau \to \tau$
Expressions	$e ::=$	$X \mid c \mid f \mid e\ e$	Type Schemes	$\sigma ::= \forall\overline{\alpha_n}.\tau$
		$\mid\ let\ X = e\ in\ e$	Assumptions	$\mathcal{A} ::= \{s_1 : \sigma_1, \ldots, s_n : \sigma_n\}$
Symbol	$s ::=$	$X \mid c \mid f$	Program rule	$r ::= f\ \bar{t} \to e$ ($\bar{t}$ linear)
Non variable symbol	$h ::=$	$c \mid f$	Program	$\mathcal{P} ::= \{r_1, \ldots, r_n\}$
Data substitution	$\theta ::=$	$[\overline{X_n/t_n}]$	Type substitution	$\pi ::= [\overline{\alpha_n/\tau_n}]$

Fig. 1. Syntax of expressions and programs

(Fapp) $f\ t_1\theta \ldots t_n\theta\ \to^{lf}\ r\theta$, if $(f\ t_1 \ldots t_n \to r) \in \mathcal{P}$

(Ffail) $f\ t_1 \ldots t_n\ \to^{lf}\ fail$, if $n = ar(f)$ and $\nexists (f\ t'_1 \ldots t'_n \to r) \in \mathcal{P}$ such that $f\ t'_1 \ldots t'_n$ and $f\ t_1 \ldots t_n$ unify

(FailP) $fail\ e\ \to^{lf}\ fail$

(LetIn) $e_1\ e_2 \to^{lf} let\ X = e_2\ in\ e_1\ X$, if e_2 is junk, active, variable application or *let* rooted, for X fresh.

(Bind) $let\ X = t\ in\ e\ \to^{lf}\ e[X/t]$

(Elim) $let\ X = e_1\ in\ e_2 \to^{lf} e_2$, if $X \notin fv(e_2)$

(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3\ \to^{lf}\ let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$, if $Y \notin fv(e_3)$

(LetAp) $(let\ X = e_1\ in\ e_2)\ e_3 \to^{lf} let\ X = e_1\ in\ e_2\ e_3$, if $X \notin fv(e_3)$

(Contx) $\mathcal{C}[e] \to^{lf} \mathcal{C}[e']$, if $\mathcal{C} \neq [\ ]$, $e \to^{lf} e'$ using any of the previous rules

Fig. 2. Higher order *let*-rewriting relation with pattern matching failure $\to^{lf}$

for data constructors may be existential, i.e., they can be of the form $\forall\overline{\alpha_n}.\overline{\tau} \to \tau'$ where $(\bigcup_{\tau_i \in \overline{\tau}} ftv(\tau_i)) \setminus ftv(\tau') \neq \emptyset$. If $(s : \sigma) \in \mathcal{A}$ we write $\mathcal{A}(s) = \sigma$. A *type substitution* π is a finite mapping from type variables to simple types $[\overline{\alpha_n/\tau_n}]$. Application of type substitutions to simple types is defined in the natural way and for type-schemes consists in applying the substitution only to their free variables. This notion is extended to set of assumptions in the obvious way. We say σ is an *instance* of σ' if $\sigma = \sigma'\pi$ for some π. A simple type τ' is a *generic instance* of $\sigma = \forall\overline{\alpha_n}.\tau$, written $\sigma \succ \tau'$, if $\tau' = \tau[\overline{\alpha_n/\tau_n}]$ for some $\overline{\tau_n}$. Finally, τ' is a *variant* of $\sigma = \forall\overline{\alpha_n}.\tau$, written $\sigma \succ_{var} \tau'$, if $\tau' = \tau[\overline{\alpha_n/\beta_n}]$ and $\overline{\beta_n}$ are fresh type variables.

3 Formal setup

3.1 Semantics

The operational semantics of our programs is based on *let*-rewriting [17], a high level notion of reduction step devised to express call-time choice. For this paper,

we have extended *let*-rewriting with two rules for managing failure of pattern matching (Fig. 2), playing a role similar to the rules for pattern matching failures in GADTs [3,23]. We write $\rightarrow^{lf}$ for the extended relation and $\mathcal{P} \vdash e \rightarrow^{lf} e'$ ($\mathcal{P} \vdash e \twoheadrightarrow^{lf} e'$ resp.) to express one step (zero or more steps resp.) of $\rightarrow^{lf}$ using the program $\mathcal{P}$. By $nf_{\mathcal{P}}(e)$ we denote the set of *normal forms* reachable from e, i.e., $nf_{\mathcal{P}}(e) = \{e' \mid \mathcal{P} \vdash e \twoheadrightarrow^{lf} e' \text{and } e' \text{ is not } \rightarrow^{lf}\text{-reducible}\}$.

The new rule (Ffail) generates a failure when no program rule can be used to reduce a function application. Notice the use of unification instead of simple pattern matching to check that the variables of the expression will not be able to match the patterns in the rule. This allows us to perform this failure test locally without having to consider the possible bindings for the free variables in the expression caused by the surrounding context. Otherwise, these should be checked in an additional condition for (Contx). Consider for instance the program $\mathcal{P}_1 = \{true \wedge X \rightarrow X, false \wedge X \rightarrow false\}$ and the expression $let\ Y = true\ in\ (Y \wedge true)$. The application $Y \wedge true$ unifies with the function rule left-hand side $true \wedge X$, so no failure is generated. If we use pattern matching as condition, a failure is incorrectly generated since neither $true \wedge X$ nor $false \wedge X$ match with $Y \wedge true$.

Finally, rule (FailP) is used to propagate the pattern matching failure when *fail* is applied to another expression.

Completing the *let*-rewriting relation of [17] has been motivated by the desire of distinguishing two kinds of failing reductions that occur in an untyped setting:

• Reductions that cannot progress because of an incomplete function definition, in the sense that the patterns of the function rules do not cover all possible cases for data constructors. A prototypical example is given by the definition *head (cons x xs)* $\rightarrow x$, where the case *head nil* is (intentionally) missing. Similar to what happens in FP systems like Haskell, we expect *(head nil)* to give raise to a failing reduction, but not to a type error. A difference is that in FP an attempt to evaluate *(head nil)* will result in a run-time error, while in FLP systems rather than an error this is a silent failure in a possible space of non-deterministic computations that is managed by backtracking. That justifies our choice of the word *fail* instead of *error*.

• Reductions that cannot progress (get *stuck*) because of a genuine type error, as happens for *junk expressions* that apply a non-functional value to some arguments (e.g. *true false*).

Our failure rules (Ffail) and (FailP) try to accomplish with the first kind of reductions. Reductions of the second kind remain stuck even with the added failure rules. As we will see in Sect. 4, this can only happen to ill-typed expressions. At the end of that section, once the type system and its formal properties have been presented, we further discuss the issues of *fail*-ended and stuck reductions.

3.2 Type derivation and inference for expressions

Both derivation and inference rules are based on those presented in [16]. Our type derivation rules for expressions (Fig. 3-a) correspond to the well-known

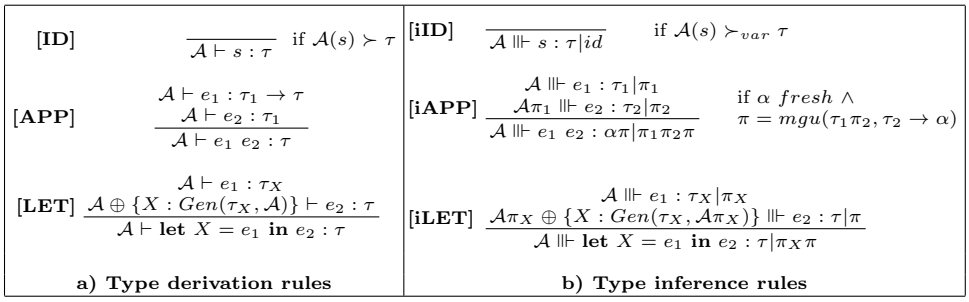

Fig. 3. Type system

variation of Damas-Milner's [5] type system with syntax-directed rules, so there is nothing essentially new here—the novelty will come from the notion of well-typed program. $Gen(\tau, \mathcal{A})$ is the closure or generalization of τ wrt. $\mathcal{A}$, which generalizes all the type variables of τ that do not appear free in $\mathcal{A}$. Formally: $Gen(\tau, \mathcal{A}) = \forall\overline{\alpha_n}.\tau$ *where* $\{\overline{\alpha_n}\} = ftv(\tau) \setminus ftv(\mathcal{A})$. We say that e is well-typed under $\mathcal{A}$, written $wt_{\mathcal{A}}(e)$, if there exists some τ such that $\mathcal{A} \vdash e : \tau$; otherwise it is ill-typed.

The type inference algorithm $\Vdash$ (Fig. 3-b) follows the same ideas as the algorithm $\mathcal{W}$ [5]. We have given the type inference a relational style to show the similarities with the typing rules. Nevertheless, the inference rules represent an algorithm that fails if no rule can be applied. This algorithm accepts a set of assumptions $\mathcal{A}$ and an expression e, and returns a simple type τ and a type substitution π. Intuitively, τ is the "most general" type which can be given to e, and π is the "most general" substitution we have to apply to $\mathcal{A}$ for deriving any type for e.

3.3 Well-typed programs

The next definition—the most important in the paper—establishes the conditions that a program must fulfil to be well-typed in our proposal:

Definition 1 (Well-typed program wrt. $\mathcal{A}$). *The program rule* $f\ t_1 \ldots t_m \to e$ *is* well-typed *wrt. a set of assumptions* $\mathcal{A}$*, written* $wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e)$*, iff:*

i) π_L *is the most general substitution such that* $wt_{(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_L}(f\ t_1 \ldots t_m)$*, and* τ_L *is the most general type derivable for* $f\ t_1 \ldots t_m$ *under the assumptions* $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_L$*.*
ii) π_R *is the most general substitution such that* $wt_{(\mathcal{A} \oplus \{\overline{X_n : \beta_n}\})\pi_R}(e)$*, and* τ_R *is the most general type derivable for* e *under the assumptions* $(\mathcal{A} \oplus \{\overline{X_n : \beta_n}\})\pi_R$*.*
iii) $\exists\pi.(\tau_L, \overline{\alpha_n\pi_L}) = (\tau_R, \overline{\beta_n\pi_R})\pi$
iv) $\mathcal{A}\pi_L = \mathcal{A}$*,* $\mathcal{A}\pi_R = \mathcal{A}$*,* $\mathcal{A}\pi = \mathcal{A}$

where $\{\overline{X_n}\} = var(f\ t_1 \ldots t_m)$ *and* $\{\overline{\alpha_n}\}, \{\overline{\beta_n}\}$ *are fresh type variables. A program* $\mathcal{P}$ *is well-typed wrt.* $\mathcal{A}$*, written* $wt_{\mathcal{A}}(\mathcal{P})$*, iff all its rules are well-typed.*

The first two points check that both right and left hand sides of the rule can have a valid type assigning *some* types for the variables, obtaining the most general types for them in both sides. The third point is the most important. It checks that the obtained most general types for the right-hand side and the variables appearing in it are more general than the ones for the left-hand side. This fact guarantees the *type preservation* property (i.e., the expression resulting after a reduction step has the same type as the original one) when applying a program rule. Moreover, this point ensures a correct management of both *skolem* constructors [14] and *opaque variables* [16], either introduced by the presence of existentially quantified constructors or higher order patterns. Finally, the last point guarantees that the set of assumptions is not modified by neither the type inference nor the matching substitution. In practice, this point holds trivially if type assumptions for program functions are closed, as it is usual. Although points i) and ii) are very declarative, we prefer a more operational behavior for Definition 1. Based on the close relationship between type derivation and inference (soundness and completeness [16]) we can replace the first two points of the definition by other equivalent ones using inference:

Definition 2 (Well-typed program wrt. $\mathcal{A}$). *The program rule* $f\ t_1 \ldots t_m \to e$ *is* well-typed *wrt. a set of assumptions* $\mathcal{A}$*, written* $wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e)$*, iff:*

i) $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t_1 \ldots t_m : \tau_L | \pi_L$
ii) $\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_R | \pi_R$
iii) $\exists \pi . (\tau_L, \overline{\alpha_n \pi_L}) = (\tau_R, \overline{\beta_n \pi_R}) \pi$
iv) $\mathcal{A}\pi_L = \mathcal{A}$, $\mathcal{A}\pi_R = \mathcal{A}$, $\mathcal{A}\pi = \mathcal{A}$

where $\{\overline{X_n}\} = var(f\ t_1 \ldots t_m)$ *and* $\{\overline{\alpha_n}\}, \{\overline{\beta_n}\}$ *are fresh type variables. A program* $\mathcal{P}$ *is well-typed wrt.* $\mathcal{A}$*, written* $wt_{\mathcal{A}}(\mathcal{P})$*, iff all its rules are well-typed.*

The equivalence between both definitions of well-typed rule is based on the following result about type derivation and inference:

Lemma 1. *π is the most general substitution that enables to derive a type for the expression e under the assumptions $\mathcal{A}$, and τ is the most general derivable type for e ($\mathcal{A}\pi \vdash e : \tau$) $\iff$ $\exists \pi', \tau'$ such that $\mathcal{A} \Vdash e : \tau' | \pi'$ and $\pi \simeq \pi'$ and $\tau \simeq \tau'$.*

The previous definition presents some similarities with the notion of *typeable* rewrite rule for Curryfied Term Rewriting Systems in [2]. In that paper the key condition is that the *principal type* for the left-hand side allows to derive the same type for the right-hand side. Besides, [2] considers intersection types and it does not provide an effective procedure to check well-typedness.

Example 1 (Well and ill-typed rules and expressions). Let us consider the following assumptions and program:

$\mathcal{A} \equiv \{\ \mathbf{z} : nat,\ \mathbf{s} : nat \to nat,\ \mathbf{true} : bool,\ \mathbf{false} : bool, \mathbf{cons} : \forall\alpha.\alpha \to [\alpha] \to [\alpha],$
$\mathbf{nil} : \forall\alpha.[\alpha],\ \mathbf{rnat} : repr\ nat,\ \mathbf{id} : \forall\alpha.\alpha \to \alpha,\ \mathbf{snd} : \forall\alpha,\beta.\alpha \to \beta \to \beta,$
$\mathbf{unpack} : \forall\alpha,\beta.(\alpha \to \alpha) \to \beta, \mathbf{eq} : \forall\alpha.\alpha \to \alpha \to bool,\ \mathbf{showNat} : nat \to [char],$
$\mathbf{show} : \forall\alpha.repr\ \alpha \to \alpha \to [char],\ \mathbf{f} : \forall\alpha.bool \to \alpha,\ \mathbf{flist} : \forall\alpha.[\alpha] \to \alpha\ \}$

$\mathcal{P} \equiv \{\ id\ X \to X,\ snd\ X\ Y \to Y,\ unpack\ (snd\ X) \to X,\ eq\ (s\ X)\ z \to false,$
$show\ rnat\ X \to showNat\ X,\ f\ true \to z,\ f\ true \to false,$
$flist\ (cons\ z\ nil) \to s\ z,\ flist\ (cons\ true\ nil) \to false\ \}$

The rules for the functions *id* and *snd* are well-typed. The function *unpack* is taken from [8] as a typical example of the type problems that HO-patterns can produce. According to Def. 2 the rule of *unpack* is not well-typed since the tuple $(\tau_L, \overline{\alpha_n \pi_L})$ inferred for the left-hand side is (γ, δ), which is not matched by the tuple (η, η) inferred as $(\tau_R, \overline{\beta_n \pi_R})$ for the right-hand side. This shows the problem of existential type variables that "escape" from the scope. If that rule was well-typed then type preservation could not be granted anymore—e.g. consider the step $unpack\ (snd\ true) \to^{lf} true$, where the type *nat* can be assigned to $unpack\ (snd\ true)$ but *true* can only have type *bool*. The rule for *eq* is well-typed because the tuple inferred for the right-hand side, $(bool, \gamma)$, matches the one inferred for the left-hand side, $(bool, nat)$. In the rule for *show* the inference obtains $([char], nat)$ for both sides of the rule, so it is well-typed.

The functions f and $flist$ show that our type system cannot be forced to accept an arbitrary function definition by generalizing its type assumption. For instance, the first rule for f is not well-typed since the type *nat* inferred for the right-hand side does not match γ, the type inferred for the left-hand side. The second rule for f is also ill-typed for a similar reason. If these rules were well-typed, type preservation would not hold: consider the step $f\ true \to^{lf} z$; $f\ true$ can have any type, in particular *bool*, but z can only have type *nat*. Concerning $flist$, its type assumption cannot be made more general for its first argument: it can be seen that there is no τ s.t. the rules for $flist$ remain well-typed under the assumption $flist : \forall\alpha.\alpha \to \tau$.

With the previous assumptions, expressions like $id\ z\ true$ or $snd\ z\ z\ true$ that lead to *junk* are ill-typed, since the symbols *id* and *snd* are applied to more expressions than the arity of their types. Notice also that although our type system accepts more expressions that may produce pattern matching failures than classical Damas-Milner, it still rejects some expressions presenting those situations. Examples of this are $flist\ z$ and $eq\ z\ true$, which are ill-typed since the type of the function prevents the existence of program rules that can be used to rewrite these expressions: $flist$ can only have rules treating lists as argument and *eq* can only have rules handling both arguments of the same type.

Def. 2 can be implemented easily. For each program rule, conditions *i*) and *ii*) use the algorithm of type inference for expressions, *iii*) is just matching, and *iv*) holds trivially in practice, as we have noticed before. We encourage the

reader to play with the implementation, made available as a web interface at `http://gpd.sip.ucm.es/LiberalTyping`.

In [16] we extended Damas-Milner types with some extra control over HO-patterns, leading to another definition of well-typed programs (we write $wt^{old}_{\mathcal{A}}(\mathcal{P})$ for that). All valid programs in [16] are still valid:

Theorem 1. *If $wt^{old}_{\mathcal{A}}(\mathcal{P})$ then $wt_{\mathcal{A}}(\mathcal{P})$.*

From this result and from the fact that typing of expressions remain the same as in [16] we can extract the following consequence: if the types of functions in a program are declared to be the types inferred in [16], then our type system behaves as the system in [16] (i.e., disregarding HO patterns, essentially as Damas-Milner). To appreciate the usefulness of the new possibilities given by the new notion with respect the old one, notice that all the examples in Sect. 5 are rejected as ill-typed by [16].

4 Properties of the type system

Our type system accepts more programs than the classical Damas-Milner type system or the type system in [16]. However, it still provides type soundness. We will follow two alternative approaches for proving type soundness. First, we will prove the theorems of *progress* and *type preservation* that play the main role in the type soundness proof for GADTs [3,23]. Finally, we will follow a syntactic approach similar to [27] and we will prove the *syntactic soundness* of our system.

The first result, *progress*, states that well-typed ground expressions are patterns or reducible by *let*-rewriting.

Theorem 2 (Progress). *If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$ and e is ground, then either e is a pattern or $\exists e'.\ \mathcal{P} \vdash e \rightarrow^{lf} e'$.*

In order to relate well-typed expressions and evaluation we need a *type preservation*—or *subject reduction*—result, stating that in well-typed programs reduction does not change types.

Theorem 3 (Type Preservation). *If $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ and $\mathcal{P} \vdash e \rightarrow^{lf} e'$, then $\mathcal{A} \vdash e' : \tau$.*

The two previous theorems extend to our more liberal typing the main results of type soundness for GADTs [3]. In order to state type soundness following a syntactic approach similar to [27] we need to define some properties about expressions:

Definition 3. *An expression e is* **stuck** *wrt. a program $\mathcal{P}$ if it is a normal form but not a pattern, and is* **faulty** *if it contains a* junk *subexpression.*

Faulty is a pure syntactic property that tries to overapproximate *stuck*. Notice that not all faulty expressions are stuck. For example, $snd\ (z\ z)\ true \rightarrow^{lf} true$ and $let\ X = true\ z\ in\ z \rightarrow^{lf} z$. However all faulty expressions are ill-typed:

Lemma 2 (Faulty Expressions are ill-typed). *If e contains a junk subexpression then there is no $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$.*

The next theorem states that all finished reductions of well-typed ground expressions end up in patterns of the same type as the original expression.

Theorem 4 (Syntactic Soundness). *If $wt_{\mathcal{A}}(\mathcal{P})$, e is ground and $\mathcal{A} \vdash e : \tau$ then: for all $e' \in nf_{\mathcal{P}}(e)$, e' is a pattern and $\mathcal{A} \vdash e' : \tau$.*

Then the fact that the evaluation of a well-typed expression cannot become *stuck* by reduction is a corollary of *syntactic soundness*. The following complementary result states that the evaluation of well-typed expressions does not produce type errors.

Theorem 5. *If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$ and e is ground, then there is no e' such that $\mathcal{P} \vdash e \twoheadrightarrow^{lf} e'$ and e' is $faulty$.*

4.1 Discussion

How strong are our results? Let us discuss it a bit, considering some interdependent factors: the rules for *failure* in Sect. 3, the liberality of our well-typedness condition, and our notion of *faulty* expression.

- **Progress and type preservation:** In his seminal paper [21] Milner includes in the semantics of a small language *'a value 'wrong', which corresponds to the detection of a failure at run-time'* (sic), proving then that *'well-typed programs don't go wrong'*. For this to be true in our language or in FP languages like Haskell, not all run-time failures should be seen as wrong. As we remarked in Sect. 3.1, some care must be taken with definitions like *head (cons x xs)* $\to x$, where there is no rule for *(head nil)*. Otherwise, progress does not hold and some well-typed expressions become stuck. A solution—implicitly or explicitly adopted in FP languages—is considering a 'well-typed completion' of the program, adding a rule like *head nil* $\to$ *error* where *error* accepts any type. With it, *(head nil)* reduces to *error* and is not wrong, but *(head true)*, which is ill-typed, is wrong and its reduction gets stuck. Our introduction of *fail*[2] through general failure rules tries to play a similar role but leads to a weaker position, since the difference between 'wrong' and 'error' partially vanishes (not totally, junk expressions still remain stuck). For instance, *(head nil)* and *(head true)* both reduce to *fail*, instead of *(head true)* getting stuck. How serious is that? Since *fail* accepts all types, this might seem a point where ill-typedness comes in hiddenly and then magically disappear by the effect of reduction to *fail*. However, *type preservation (Th. 3)* comes to our rescue: since it holds step-by-step, no reduction $e \to^* fail$ starting with a well-typed e can pass through the ill-typed *(head true)* as intermediate (sub)-expression. Nevertheless, why didn't we simply preclude formally the reduction *(head true)* $\to$ *fail* by using the above sketched well-typed completions? The main reason is that in our setting they are more complex to

[2] We explained in Sect. 4 why we prefer the name *'fail* instead of *'error'* for the FLP setting.

formulate than failure rules, specially in presence of program rules with HO-patterns, because there could be an infinite number of 'missing' HO-patterns of the same type.

- **Liberality:** All typed languages present the problem of accepting as well-typed some expressions that one might prefer to detect and reject at compile time. In our liberal system the risk is higher. Consider the example in Sect. 1 where *size* had declared type $\forall\alpha.\alpha \to nat$. For any well-typed e, the expression *size e* is also well-typed, even if e's type is not considered in the definition of *size*; for instance, *size (true,false)* is a well-typed expression reducing to *fail*. This is consistent with the liberality of our system, since the definition of *size* could perfectly have included a rule for computing sizes of pairs. Hence, for our system, this can only seen as a pattern matching failure due to a missing case in the definition, in all senses similar to the case of *(head nil)*. This situation, that can be appreciated as a weakness, is further discussed in Sect. 6 in connection to type classes and GADT's.
- **Syntactic soundness and faulty expressions:** Th. 4 and 5 do not add too much information to progress and type preservation, from which they are easy consequences. Th. 5 is indeed a weaker safety criterion, due to the fact that our notion of *faulty* is weaker than the analogous notion in [27], which covers more exactly the set of ill-typed expressions for the language in that paper. In our case faulty expressions only capture the presence of junk, which by no means is the only source of ill-typedness. For instance, with the usual type assumptions, the expressions *(head true)* or *(eq true z)* are ill-typed but not faulty (see Sect. 3.3 for more expamples). Th. 5 says nothing about them; it is type preservation who ensures that those expressions will not occur in any reduction starting in a well-typed expression. Despite of its weakness compared to type preservation, Th. 5 still contains no trivial information. Certainly, the property that a given expression is faulty is trivial to check by counting arguments. However, this is not the same as considering the property of whether a given expression will become faulty or not during reduction, a typically undecidable property of which our type system serves as a approximation. For example, consider a function g with declared type $\forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta$, defined by the (well-typed) program rule $g\ H\ X \to H\ X$, and the expression *(g true false)*, which is not faulty, but reduces to the faulty *(true false)*. Our type system detects and avoids that because the non-faulty expression *(g true false)* is detected as ill-typed. Would have *(g true false)* been well-typed, Th. 5 would have guaranteed that it could not reduce to a faulty expression.

5 Examples

In this section we present some examples showing the flexibility achieved by our type system. They are written in two parts: a set of assumptions $\mathcal{A}$ over constructors and functions and a set of program rules $\mathcal{P}$. In the examples we

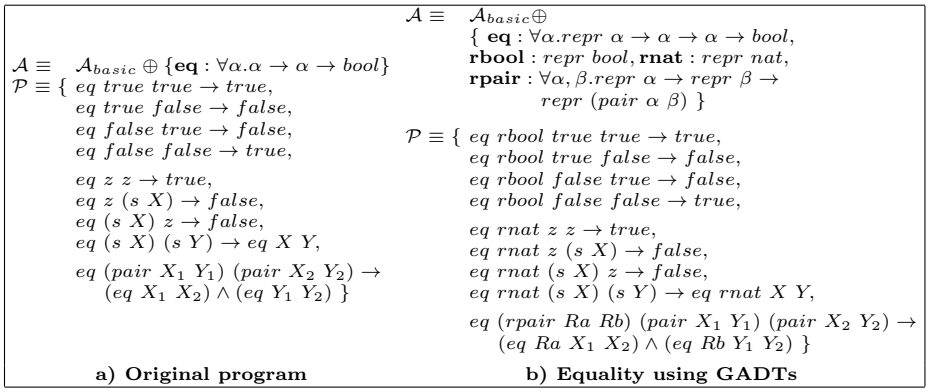

Fig. 4. Type-indexed equality

consider the following initial set of assumptions:

$\mathcal{A}_{basic} \equiv \{\mathbf{true}, \mathbf{false} : bool,\ \mathbf{z} : nat,\ \mathbf{s} : nat \to nat,\ \mathbf{cons} : \forall\alpha.\alpha \to [\alpha] \to [\alpha],$
$\mathbf{nil} : \forall\alpha.[\alpha],\ \mathbf{pair} : \forall\alpha,\beta.\alpha \to \beta \to pair\ \alpha\ \beta,\ \mathbf{key} : \forall\alpha.\alpha \to (\alpha \to nat) \to key,$
$\wedge, \vee : bool \to bool \to bool,\ \mathbf{snd} : \forall\alpha,\beta.\alpha \to \beta \to \beta, \}$

5.1 Type-indexed functions

Type-indexed functions (in the sense appeared in [12]) are functions that have a particular definition for each type in a certain family. The function *size* of Sect. 1 is an example of such a function. A similar example is given in Fig. 4-a, containing the code for an equality function which only operates with booleans, natural numbers and pairs.

An interesting point is that we do not need a type representation as an extra argument of this function as we would need in a system using GADTs [3,12]. In these systems the pattern matching on the GADT induces a type refinement, allowing the rule to have a more specific type than the type of the function. In our case this flexibility resides in the notion of well-typed rule. Then a type representation is not necessary because the arguments of each rule of *eq* already force the type of the left-hand side and its variables to be more specific (or the same) than the inferred type for the right-hand side. The absence of type representations provides simplicity to rules and programs, since extra arguments imply that all functions using *eq* direct or indirectly must be extended to accept and pass these type representations. In contrast, our rules for *eq* (extended to cover all constructed types) are the standard rules defining strict equality that one can find in FLP papers (see e.g. [9]), but that cannot be written directly in existing systems like TOY or Curry, because they are ill-typed according to Damas-Milner types.

We stress also the fact that the program of Fig. 4-a would be rejected by systems supporting GADTs [3,24], while the encoding of equality using GADTs as type representations in Fig. 4-b is also accepted by our type system.

Another interesting point is that we can handle equality in a quite fine way, much more flexible than in TOY or Curry, where equality is a *built-in* that proceeds structurally as in Fig. 4-a. With our proposed type system programmers can define structural equality as in Fig. 4-a for some types, choose another behavior for others, and omitting the rules for the cases they do not want to handle. Moreover, the type system protects against unsafe definitions, as we explain now: it is known [8] that in the presence of HO-patterns[3] structural equality can lead to the problem of *opaque decomposition*. For example, consider the expression $eq\ (snd\ z)\ (snd\ true)$. It is well-typed, but after a decomposition step using the structural equality we obtain $eq\ z\ true$, which is ill-typed. Different solutions have been proposed [8], but all of them need fully type-annotated expressions at run time, which penalizes efficiency. With the proposed type system that overloading at run time is not necessary since this problem of opaque decomposition is handled statically at compile time: we simply cannot write equality rules leading to opaque decomposition, because they are rejected by the type system. This happens with the rule $eq\ (snd\ X)\ (snd\ Y) \to eq\ X\ Y$, which will produce the previous problematic step. It is rejected because the inferred type for the right-hand side and its variables X and Y is $(bool, \gamma, \gamma)$, which is more specific than the inferred in the left-hand side $(bool, \alpha, \beta)$.

5.2 Existential types, opacity and HO patterns

Existential types [14] appear when type variables in the type of a constructor do not occur in the final type. For example the constructor $key : \forall\alpha.\alpha \to (\alpha \to nat) \to key$ has an existential type, since α does not appear in the final type key. In functional logic languages, however, HO-patterns can introduce the same opacity as constructors with existential type. A prototypical example is $snd\ X$: we know that X has some type, but we cannot know anything about it from the type $\beta \to \beta$ of the expression. In [16] a type system managing the opacity of HO-patterns is proposed. The program below shows how the system presented here generalizes [16], accepting functions that were rejected there (e.g. $idSnd$) and also supporting constructors with existential type (e.g. $getKey$):

$$\mathcal{A} \equiv \mathcal{A}_{basic} \oplus \{\ \mathbf{getKey} : key \to nat,\ \mathbf{idSnd} : \forall\alpha,\beta.(\alpha \to \alpha) \to (\beta \to \beta)\ \}$$
$$\mathcal{P} \equiv \{\ getKey\ (key\ X\ F) \to F\ X,\ idSnd\ (snd\ X) \to snd\ X\ \}$$

Another remarkable example is given by the well-known translation of higher-order programs to first-order programs often used as a stage of the compilation of functional logic programs (see e.g. [17,1]). In short, this translation introduces a new function symbol @, adds calls to @ in some points in the program, and adds appropriate rules for evaluating it. This latter aspect is interesting here, since the rules are not Damas-Milner typeable. The following program contains the @-rules for a concrete example involving the data constructors z, s, nil, $cons$

[3] This situation also appears with first order patterns containing data constructors with existential types.

and the functions *length*, *append* and *snd* with the usual types.

$$\mathcal{A} \equiv \mathcal{A}_{basic} \oplus \{\ \mathbf{length} : \forall\alpha.[\alpha] \to nat, \mathbf{append} : \forall\alpha.[\alpha] \to [\alpha] \to [\alpha],\\ \mathbf{add} : nat \to nat \to nat, @ : \forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta\ \}$$

$$\mathcal{P} \equiv \{\ s\ @\ X \to s\ X, cons\ @\ X \to cons\ X, (cons\ X)\ @\ Y \to cons\ X\ Y,\\ append\ @\ X \to append\ X, (append\ X)\ @\ Y \to append\ X\ Y,\\ snd\ @\ X \to snd\ X, (snd\ X)\ @\ Y \to snd\ X\ Y, length\ @\ X \to length\ X\ \}$$

These rules use HO-patterns, which is a cause of rejection in most systems. Even if HO patterns were allowed, the rules for @ would be rejected by a Damas-Milner type system, no matter if extended to support existential types or GADTs. However using Def. 3.1 they are all well-typed, provided we declare @ to have the type $@ : \forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta$. Because of all this, the @-introduction stage of the FLP compilation process can be considered as a source to source transformation, instead of a hard-wired step.

5.3 Generic functions

According to a strict view of genericity, the functions *size* and *eq* in Sect. 1 and 5.1 resp. are not truly generic. We have a definition for each type, instead of one 'canonical' definition to be used by each concrete type. However we can achieve this by introducing a 'universal' data type over which we define the function (we develop the idea for *size*), and then use it for concrete types via a conversion function.

This can be done by using GADTs to represent uniformly the applicative structure of expressions (for instance, the *spines* of [12]), by defining *size* over that uniform representations, and then applying it to concrete types via conversion functions. Again, we can also offer a similar but simpler alternative. A uniform representation of constructed data can be achieved with a data type *data univ = c nat [univ]* where the first argument of *c* is for numbering constructors, and the second one is the list of arguments of a constructor application. A universal *size* can be defined as *usize (c _ Xs) → s (sum (map usize Xs))* using some functions of Haskell's prelude. Now, a generic *size* can be defined as $size \to usize \cdot toU$, where *toU* is a conversion function with declared type *toU* $: \forall\alpha.\alpha \to univ$

toU true → *c z []* *toU false* → *c (s z) []*
toU z → *c* (s^2 *z) []* *toU (s X)* → *c* (s^3 *z) [toU X]*
toU [] → *c* (s^4 *z) []* *toU (X:Xs)* → *c* (s^5 *z) [toU X,toU Xs]*

(s^i abbreviates iterated *s*'s). This *toU* function uses the specific features of our system. It is interesting also to remark that in our system the truly generic rule $size \to usize \cdot toU$ can coexist with the type-indexed rules for *size* of Sect. 1. This might be useful in practice: one can give specific, more efficient definitions

for some concrete types, and a generic default case via toU conversion for other types[4].

Admittedly, the type *univ* has less representation power than the spines of [12], which could be a better option in more complex situations. Nevertheless, notice that since spines are based on GADTs, they are also supported by our system.

6 Discussion

We further discuss here some positive and negative aspects of our type system.
Simplicity. Our well-typedness condition, which adds only one simple check for each program rule to standard Damas-Milner inference, is much easier to integrate in existing FLP systems than, for instance, type classes (see [19] for some known problems for the latter).
Liberality (continued from Sect. 4): we recall the example of *size*, where our system accepts as well-typed *(size e)* for any well-typed e. Type classes impose more control: *size* e is only accepted if e has a type in the class *Sizeable*. There is a burden here: you need a class for each generic function, or at least for each range of types for which a generic function exists; therefore, the number of class instance declarations for a given type can be very high. GADTs are in the middle way. At a first sight, it seems that the types to which *size* can be applied are perfectly controlled because only *representable* types are permitted. The problem, as with classes, comes when considering other functions that are generic but for other ranges of types. Now, there are two options: either you enlarge the family of representable functions, facing up again the possibility of applying *size* to unwanted arguments, or you introduce a new family of representation types, which is a programming overhead, somehow against genericity.
Need of type declarations. In contrast to Damas & Milner system, where types can be inferred and need not to be declared, our definition of well-typed program (Def. 2) assumes an explicit type declaration for each function. This happens also with other well-known type features, like polymorphic recursion, arbitrary-rank polymorphism or GADTs [3,24]. Moreover, programmers usually declare the types of functions as a way of documenting programs. Notice also that type inference for functions would be a difficult task since functions, unlike expressions, do not have *principal types*. Consider for instance the rule $not\ true \to false$. All the possible types for the *not* function are $\forall\alpha.\alpha \to \alpha$, $\forall\alpha.\alpha \to bool$ and $bool \to bool$ but none of them is most general.
Loss of parametricity. In [26] one of the most remarkable applications of type systems was developed. The main idea there is to derive "free theorems" about the equivalence of functional expressions by just using the types of some of its constituent functions. These equivalences express different distribution properties, based on Reynold's abstraction theorem there recasted as "the parametric-

[4] For this to be really practical in FLP systems, where there is not a 'first-fit' policy for pattern matching in case of overlapping rules, a specific syntactic construction for 'default rule' would be needed.

ity theorem", which basically exploits the fact that a function cannot inspect the values of argument subexpressions with a polymorphic variable as type. Parametricity was originally developed for the polymorphic λ-calculus, so free theorems have to be weakened with additional conditions in order to accomodate them to practical languages like Haskell, as their original formulations are false in the presence of unbounded recursion, partial functions or impure features like seq [26,13].
With our type system parametricity is lost, because functions are allowed to inspect any argument subexpression, as seen in the *size* function from page 2. This has a limited impact in the FLP setting, since it is known that non-determinism and narrowing—not treated in the present work but standard in FLP systems—not only breaks free theorems but also equational rules for concrete functions that hold for Haskell, like $(filter\ p) \circ (map\ h) \equiv (map\ h) \circ (filter\ (p \circ h))$ [4].

7 Conclusions

Starting from a simple type system, essentially Damas-Milners's one, we have proposed a new notion of well-typed functional logic program that exhibits interesting properties: simplicity; enough expressivity to achieve existential types, GADTs and to open new possibilities to genericity; good formal properties (type soundness, protection against unsafe use of HO patterns). Regarding the practical interest of our work, we stress the fact that no existing FLP system supports any of the examples in Sect. 5, in particular the examples of the *equality*—where known problems of *opaque decomposition* [8] can be addressed—and *apply* functions, which play important roles in the FLP setting. Moreover, our work greatly improves our previous results [16] about safe uses of HO patterns. However, considering also the weaknesses discussed in Sect. 6 suggests that a good option in practice could be a partial adoption of our system, not attempting to replace standard type inference, type classes or GADTs, but rather complementing them.

We find suggestive to think of the following future scenario for our system TOY: a typical program will use standard type inference except for some concrete definitions where it is annotated that our new liberal system is adopted instead. In addition, adding type classes to the languages is highly desirable; then the programmer can choose the feature—ordinary types, classes, GADTs or our more direct generic functions—that best fits his needs of genericity and/or control in each specific situation. We have some preliminary work [20] exploring the use of our type-indexed functions to implement type classes in FLP, with some advantages over the classical dictionary-based technology.

Apart from the implementation work, to realize that vision will require further developments of our present work:

• A precise specification of how to mix different typing conditions in the same program and how to translate type classes into our generic functions.

• Despite of the lack of principal types, some work on type inference can be done, in the spirit of [24].

• Combining our genericity with the existence of modules could require adopting *open* types and functions [15].

• Narrowing, which poses specific problems to types, should be also considered.

Acknowledgments We thank Philip Wadler and the rest of reviewers for their useful comments.

References

1. Antoy, S., Tolmach, A.P.: Typed higher-order narrowing without higher-order strategies. In: Proc. FLOPS'99, Springer LNCS 1722, pp. 335–353, 1999.
2. van Bakel, S., Fernández, M.: Normalization Results for Typeable Rewrite Systems. Information and Computation 133(2), pp. 73–116, 1997.
3. Cheney, J., Hinze, R.: First-class phantom types. Tech. Rep. TR2003-1901, Cornell University, 2003.
4. Christiansen, J., Seidel, D., Voigtländer, J.: Free theorems for functional logic programs. In: Proc. PLPV '10, pp. 39–48. ACM, 2010.
5. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proc. POPL'82, pp. 207–212. ACM, 1982.
6. González-Moreno, J.C., Hortalá-González, T., López-Fraguas, F., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. Journal of Logic Programming 40(1), pp. 47–87, 1999.
7. González-Moreno, J., Hortalá-González, M., Rodríguez-Artalejo, M.: A higher order rewriting logic for functional logic programming. In: Proc. ICLP'97, pp. 153–167. MIT Press, 1997.
8. Gonzalez-Moreno, J.C., Hortala-Gonzalez, M.T., Rodriguez-Artalejo, M.: Polymorphic types in functional logic programming. Journal of Functional and Logic Programming 2001(1), 2001.
9. Hanus, M.: Multi-paradigm declarative languages. In: Proc. ICLP'07, Springer LNCS 4670, pp. 45–75, 2007.
10. Hanus (ed.), M.: Curry: An integrated functional logic language (version 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, 2006.
11. Hinze, R.: Generics for the masses. J. Funct. Program. 16(4-5), pp. 451–483, 2006.
12. Hinze, R., Löh, A.: Generic programming, now!. In: Revised Lectures SSDGP'06, Springer LNCS 4719, pp. 150–208, 2007.
13. Hudak, P., Hughes, J., Jones, S.P., Wadler, P.: A History of Haskell: being lazy with class. In: Proc. HOPL III, pp. 12-1–12-55. ACM, 2007.
14. Läufer, K., Odersky, M.: Polymorphic type inference and abstract data types. ACM Transactions on Programming Languages and Systems 16. ACM, 1994.
15. Löh, A., Hinze, R.: Open data types and open functions. In: Proc. PPDP '06, pp. 133–144. ACM, 2006.
16. López-Fraguas, F.J., Martin-Martin, E., Rodríguez-Hortalá, J.: New results on type systems for functional logic programming. In: WFLP'09 Revised Selected Papers, Springer LNCS 5979, pp. 128–144, 2010.
17. López-Fraguas, F., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: Rewriting and call-time choice: the HO case. In: Proc. FLOPS'08, Springer LNCS 4989, pp. 147–162, 2008.
18. López-Fraguas, F., Sánchez-Hernández, J.: $\mathcal{TOY}$: A multiparadigm declarative system. In: Proc. RTA'99, Springer LNCS 1631, pp. 244–247, 1999.

19. Lux, W.: Adding haskell-style overloading to curry. In: Workshop of Working Group 2.1.4 of the German Computing Science Association GI. pp. 67–76, 2008.
20. Martin-Martin, E.: Implementing type classes using type-indexed functions. To appear in TPF'10, available at `http://gpd.sip.ucm.es/enrique/publications/implementingTypeClasses/implementingTypeClasses.pdf`.
21. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348–375 (1978)
22. Moreno-Navarro, J.J., Mariño, J., del Pozo-Pietro, A., Herranz-Nieva, Á., García-Martín, J.: Adding type classes to functional-logic languages. In: Proc. APPIA-GULP-PRODE'96. pp. 427–438, 1996.
23. Peyton Jones, S., Vytiniotis, D., Weirich, S.: Simple unification-based type inference for GADTs. Tech. Rep. MS-CIS-05-22, Univ. Pennsylvania, 2006.
24. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proc. ICFP '09, pp. 341–352. ACM, 2009.
25. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proc. POPL'89, pp. 60–76. ACM, 1989.
26. Wadler, P.: Theorems for free! In: Proc. FPCA'89, pp. 347–359. ACM, 1989.
27. Wright, A.K., Felleisen, M.: A Syntactic Approach to Type Soundness. Information and Computation 115, pp. 38–94, 1992.

A Proofs and auxiliary results

We first present some notions used in the proofs.

a) For any type substitution π its *domain* is defined as $dom(\pi) = \{\alpha \mid \alpha\pi \neq \alpha\}$; and the *variable range* of π is $\bigcup_{\alpha \in dom(\pi)} ftv(\alpha\pi)$

b) Provided the domains of two type substitutions π_1 and π_2 are disjoints, the *simultaneous composition* $(\pi_1 + \pi_2)$ is defined as:

$$\alpha(\pi_1 + \pi_2) = \begin{cases} \alpha\pi_1 & \text{if } \alpha \in dom(\pi_1) \\ \alpha\pi_2 & \text{otherwise} \end{cases}$$

c) If A is a set of type variables, the *restriction* of a substitution π to A $(\pi|_A)$ is defined as:

$$\alpha(\pi|_A) = \begin{cases} \alpha\pi & \text{if } \alpha \in A \\ \alpha & \text{otherwise} \end{cases}$$

We use $\pi|_{\setminus A}$ as an abbreviation of $\pi|_{\mathcal{TV} \setminus A}$

A.1 Auxiliary results

Theorem 6 shows that the type and substitution found by the inference are correct, i.e., we can build a type derivation for the same type if we apply the substitution to the assumptions.

Theorem 6 (Soundness of $\Vdash$). $\mathcal{A} \Vdash e : \tau | \pi \Longrightarrow \mathcal{A}\pi \vdash e : \tau$

Theorem 7 expresses the completeness of the inference process. If we can derive a type for an expression applying a substitution to the assumptions, then inference will succeed and will find a type and a substitution which are the most general ones.

Theorem 7 (Completeness of $\Vdash$ wrt. $\vdash$). *If* $\mathcal{A}\pi' \vdash e : \tau'$ *then* $\exists \tau, \pi, \pi''. \mathcal{A} \Vdash e : \tau | \pi \wedge \mathcal{A}\pi\pi'' = \mathcal{A}\pi' \wedge \tau\pi'' = \tau'$.

The following theorem shows some useful properties of the typing relation $\vdash$, used in the proofs.

Theorem 8 (Properties of the typing relation).

a) If $\mathcal{A} \vdash e : \tau$ *then* $\mathcal{A}\pi \vdash e : \tau\pi$, *for any* π
b) Let s *be a symbol not appearing in* e. *Then* $\mathcal{A} \vdash e : \tau \Longleftrightarrow \mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$.
c) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash e : \tau$ *and* $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$ *then* $\mathcal{A} \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$.

Proof. The proof of Theorems 6, 7 and 8 appears in Enrique Martin-Martin's master thesis[5].

□

Remark 1. If $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ and $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash e : \tau' | \pi$ then we can assume that $\mathcal{A}\pi = \mathcal{A}$.

[5] `http://gpd.sip.ucm.es/enrique/publications/master/masterThesis.pdf`

A.2 Equivalence of Def. 1 and 2

Lemma 3. *π is the most general substitution that enables to derive a type for the expression e under the assumptions $\mathcal{A}$, and τ is the most general derivable type for e ($\mathcal{A}\pi \vdash e : \tau$) $\iff \exists \pi', \tau'$ such that $\mathcal{A} \Vdash e : \tau' | \pi'$ and $\pi \simeq \pi'$ and $\tau \simeq \tau'$.*

Proof. Straightforward based on soundness (Th. 6) and completeness (Th. 7) of the inference relation.

A.3 Proof of Theorem 1

Proof. In [16] and this paper the definition of well-typed program proceeds rule by rule, so we only have to prove that if $wt^{old}_{\mathcal{A}}(f\ t_1 \ldots t_n \to e)$ then $wt_{\mathcal{A}}(f\ t_1 \ldots t_n \to e)$. For the sake of conciseness we will consider functions with just one argument: $f\ t \to e$. Since patterns are linear (all the variables are different) the proof for functions with more arguments follows the same ideas.

From $wt^{old}_{\mathcal{A}}(f\ t \to e)$ we know that $\mathcal{A} \vdash^{\bullet} \lambda t.e : \tau'_t \to \tau'_e$, being $\tau'_t \to \tau'_e$ a variant of $\mathcal{A}(f)$. Then we have a type derivation of the form:

$$[\Lambda]\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau'_t \\ \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau'_e\end{array}}{\mathcal{A} \vdash \lambda t.e : \tau'_t \to \tau'_e}$$

and we know that $critVar_{\mathcal{A}}(\lambda t.e) = \emptyset$, i.e., that $opaqueVar_{\mathcal{A}}(t) \cap fv(e)) = \emptyset$. We want to prove that:

a) $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t : \tau_L | \pi_L$
b) $\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_R | \pi_R$
c) $\exists \pi.(\tau_R, \overline{\beta_n \pi_R})\pi = (\tau_L, \overline{\alpha_n \pi_L})$
d) $\mathcal{A}\pi_L = \mathcal{A}$, $\mathcal{A}\pi_R = \mathcal{A}$, $\mathcal{A}\pi = \mathcal{A}$

By the type derivation of t and Theorem 7 we obtain the type inference

$$\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t | \pi_t$$

and there exists a type substitution π''_t such that $\tau_t \pi''_t = \tau'_t$ and $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \pi''_t = \mathcal{A} \oplus \{\overline{X_n : \tau_n}\}$, i.e., $\mathcal{A}\pi_t\pi''_t = \mathcal{A}$ and $\alpha_i \pi_t \pi''_t = \tau_i$. Moreover, from $critVar_{\mathcal{A}}(\lambda t.e) = \emptyset$ we know that for every data variable $X_i \in fv(e)$ then $ftv(\alpha_i \pi_t) \subseteq ftv(\tau_t)$. Then we can build the type inference for the application $f\ t$:

$$[\mathbf{i}\Lambda]\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f : \tau'_t \to \tau'_e | id \\ (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})id \Vdash t : \tau_t | \pi_t\end{array}}{\mathbf{a)}\ \mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t : \gamma \pi_g | \pi_t \pi_g}$$

By Remark 1 we are sure that $\mathcal{A}\pi_t = \mathcal{A}$. Since $\tau'_t \to \tau'_e$ is a variant of $\mathcal{A}(f)$ we know that it contains only free type variables in $\mathcal{A}$ or fresh variables, so $(\tau'_t \to \tau'_e)\pi_t = \tau'_t \to \tau'_e$. In order to complete the type inference we need to create

a unifier π_u for $(\tau'_t \to \tau'_e)\pi_t$ and $\tau_t \to \gamma$, being γ a fresh type variable. Notice that by Theorem 7 we know that $\mathcal{A}\pi_t\pi''_t = \mathcal{A}$ and by Remark 1 $\mathcal{A}\pi_t = \mathcal{A}$, so $\mathcal{A}\pi''_t = \mathcal{A}$. Since $\tau'_t \to \tau'_e$ contains only type variables which are free in $\mathcal{A}$ or fresh type variables generated during the inference, π''_t will not affect it. Defining π_u as $\pi''_t|_{ftv(\tau_t)} + [\gamma/\tau'_e]$ we have an unifier, since:

$$
\begin{array}{ll}
\underline{(\tau'_t \to \tau'_e)\pi_t\pi_u} & \pi_t \text{ does not affect } \tau'_t \to \tau'_e \\
= \underline{(\tau'_t \to \tau'_e)\pi_u} & \gamma \notin ftv(\tau'_t \to \tau'_e) \\
= (\tau'_t \to \tau'_e)\underline{\pi''_t|_{ftv(\tau_t)}} & \pi''_t|_{ftv(\tau_t)} \text{ does not affect } \tau'_t \to \tau'_e \\
= \tau'_t \to \underline{\tau'_e} & \text{definition of } \pi_u \\
= \underline{\tau'_t} \to \gamma\pi_u & \text{Theorem 7: } \tau_t\pi''_t = \tau'_t \\
= \underline{\tau_t\pi''_t|_{ftv(\tau_t)}} \to \gamma\pi_u & \gamma \notin ftv(\tau_t) \\
= \underline{\tau_t\pi_u \to \gamma\pi_u} & \text{application of substitution} \\
= (\tau_t \to \gamma)\pi_u &
\end{array}
$$

Moreover, it is clear that π_u is a *most general unifier* of $(\tau'_t \to \tau'_e)\pi_t$ and $\tau_t \to \gamma$, so $\pi_g \equiv \pi''_t|_{ftv(\tau_t)} + [\gamma/\tau'_e]$.

By Theorem 7 and the type derivation for e we obtain the type inference:

$$\mathbf{b)}\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_e|\pi_e$$

and there exists a type substitution π''_e such that $\tau_e\pi''_e = \tau'_e$ and $(\mathcal{A}\oplus\{\overline{X_n : \beta_n}\})\pi_e\pi''_e = \mathcal{A} \oplus \{\overline{X_n : \tau_n}\}$, i.e., $\mathcal{A}\pi_e\pi''_e = \mathcal{A}$ and $\beta_i\pi_e\pi''_e = \tau_i$. By Remark 1 we also know that $\mathcal{A}\pi_e = \mathcal{A}$, so $\mathcal{A}\pi''_e = \mathcal{A}$.

To prove **c)** we need to find a type substitution π such that $(\tau_e, \overline{\beta_n\pi_e})\pi = (\gamma\pi_g, \overline{\alpha_n\pi_t\pi_g})$. Let I be the set containing the indexes of the data variables in t which appear in $fv(e)$ and N its complement. We can define the substitution π as the simultaneous composition:

$$\pi \equiv \pi''_e|_{\setminus\{\beta_i|i\in N\}} + \{\beta_i/\alpha_i\pi_t\pi_g | i \in N\}$$

This substitution is well defined because the domains of the two substitutions are disjoint. The first component is the substitution π''_e restricted to the variables which appear in its domain but not in $\{\beta_i|i \in N\}$, while the domain of the second component contains only the variables $\{\beta_i|i \in N\}$. Notice that the data variables in $\{X_i|i \in N\}$ do not occur in $fv(e)$ so they are not involved in the type inference for e. Therefore the type variables in $\{\beta_i|i \in N\}$ do not appear in $ftv(\tau_e)$, $dom(\pi_e)$ or $vRan(\pi_e)$. With this substitution π the equality $(\tau_e, \overline{\beta_n\pi_e})\pi = (\gamma\pi_g, \overline{\alpha_n\pi_t\pi_g})$ holds because:

- Since $\tau_e\pi''_e = \tau'_e$ and the type variables in $\{\beta_i|i \in N\}$ do not occur in $ftv(\tau_e)$ we know that $\tau_e\pi = \tau_e\pi''_e|_{\setminus\{\beta_i|i\in N\}} = \tau_e\pi''_e = \tau'_e = \gamma\pi_g$.
- We know that the variables in $\{X_i|i \in I\}$ cannot be opaque in t, so $ftv(\alpha_i\pi_t) \subseteq ftv(\tau_t)$ for every $i \in I$ and $\alpha_i\pi_t\pi_g = \alpha_i\pi_t\pi''_t|_{ftv(\tau_t)} = \tau_i$ for those variables. Since the type variables $\{\beta_i|i \in N\}$ do not occur in $vRan(\pi_e)$ then $\beta_i\pi_e\pi = \beta_i\pi_e\pi''_e|_{\setminus\{\beta_i|i\in N\}} = \beta_i\pi_e\pi''_e = \tau_i = \alpha_i\pi_t\pi_g$ for every $i \in I$.

– Since the type variables $\{\beta_i | i \in N\}$ do not occur in $dom(\pi_e)$ then $\beta_i \pi_e \pi = \beta_i \pi = \alpha_i \pi_t \pi_g$ for every $i \in N$.

Finally, we have to prove that **d)** $\mathcal{A}\pi_t\pi_g = \mathcal{A}$, $\mathcal{A}\pi_e = \mathcal{A}$ and $\mathcal{A}\pi = \mathcal{A}$. For the first case we already know that $\mathcal{A}\pi_t = \mathcal{A}$ and $\mathcal{A}\pi''_t = \mathcal{A}$. Since π_g is defined as $\pi''_t|_{ftv(\tau_t)} + [\gamma/\tau'_e]$ and γ is a fresh type variable not appearing in $ftv(\mathcal{A})$ then $\mathcal{A}\pi_t\pi_g = \mathcal{A}\pi_g = \mathcal{A}\pi''_t|_{ftv(\tau_t)} = \mathcal{A}$. For the second case, $\mathcal{A}\pi_e = \mathcal{A}$ holds using Remark 1. For the last case we know that $\mathcal{A}\pi''_e = \mathcal{A}$. Since π is defined as $\pi''_e|_{\setminus\{\beta_i | i \in N\}} + \{\beta_i/\alpha_i\pi_t\pi_g | i \in N\}$ and no type variable β_i appears in $ftv(\mathcal{A})$ (they are fresh type variables) then $\mathcal{A}\pi = \mathcal{A}\pi''_e = \mathcal{A}$.

□

A.4 Proof of Theorem 2: Progress

Proof. By induction over the structure of e

Base case

X) This cannot happen because e is ground.
$c \in CS^n$) Then c is a pattern, regardless of its arity n. This case covers $e \equiv \mathit{fail}$.
$f \in FS^n$) Depending on n there are two cases:
- $n > 0$) Then f is a partially applied function symbols, so it is a pattern.
- $n = 0$) If there is a rule $(f \to r) \in \mathcal{P}$ then we can apply rule (Fapp), so $\mathcal{P} \vdash s \to^{lf} r$. Otherwise there is not any rule $(l \to r) \in \mathcal{P}$ such that l and f unify, so we can apply the rule for the matching failure (Ffail) obtaining $\mathcal{P} \vdash s \to^{lf} \mathit{fail}$.

Inductive Step

$e_1\ e_2$) From the premises we know that there is a type derivation:

$$[\mathbf{APP}]\frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

Both e_1 and e_2 are well-typed and ground. If e_1 is not a pattern, by the Induction Hypothesis we have $\mathcal{P} \vdash e_1 \to^{lf} e'_1$ and using the (Contx) rule we obtain $\mathcal{P} \vdash e_1\ e_2 \to^{lf} e'_1\ e_2$. If e_2 is not a pattern we can apply the same reasoning. Therefore we only have to treat the case when both e_1 and e_2 are patterns. We make a distinction over the structure of the pattern e_1:

- X) This cannot happen because e_1 is ground.
- $c\ t_1 \ldots t_n$ with $c \in CS^m$ and $n \leq m$) Depending on m and n we distinguish two cases:
  - $n < m$) Then $e_1\ e_2$ is $c\ t_1 \ldots t_n\ e_2$ with $n + 1 \leq m$, which is a pattern.
  - $n = m$)
    - If $c = \mathit{fail}$ then $m = n = 0$, so we have the expression $\mathit{fail}\ e_2$. In this case we can apply rule (FailP), so $\mathcal{P} \vdash \mathit{fail}\ e_2 \to^{lf} \mathit{fail}$.

  - ∗ Otherwise $e_1\ e_2$ is $c\ t_1 \ldots t_n\ e_2$ with $n+1 > m$, which is *junk*. This cannot happen because $\mathcal{A}$ is coherent and $\mathcal{A} \vdash e_1\ e_2 : \tau$, and Lemma 4 states that *junk* expressions cannot be well-typed wrt. a coherent set of assumptions.
- $f\ t_1 \ldots t_n$ with $c \in FS^m$ and $n < m$) Depending on m and n we distinguish two cases:
  - $n+1 < m$) Then $e_1\ e_2$ is $f\ t_1 \ldots t_n\ e_2$ which is a partially applied function symbol, i.e., a pattern.
  - $n+1 = m$) Then $e_1\ e_2$ is $f\ t_1 \ldots t_n\ e_2$. If there is a rule $(l \to r) \in \mathcal{P}$ such that $l\theta = f\ t_1 \ldots t_n\ e_2$ then we can apply rule (Fapp), so $\mathcal{P} \vdash e_1\ e_2 \to^{lf} r\theta$. If such a rule does not exist, then there is not any rule $(l' \to r') \in \mathcal{P}$ such that l' and $f\ t_1 \ldots t_n\ e_2$ unify. Notice that since $f\ t_1 \ldots t_n\ e_2$ is ground it does not have variables, so in this case pattern matching and unification are equivalent. Therefore we can apply the rule for the matching failure (Ffail) obtaining $\mathcal{P} \vdash e_1\ e_2 \to^{lf} \mathit{fail}$.

$let\ X = e_1\ in\ e_2$) From the premises we know that there is a type derivation:

$$[\mathbf{LET}]\frac{\mathcal{A} \vdash e_1 : \tau_X \qquad \mathcal{A} \oplus \{X : Gen(\tau_X, \mathcal{A})\} \vdash e_2 : \tau}{\mathcal{A} \vdash let\ X = e_1\ in\ e_2 : \tau}$$

There are two case whether e_1 is a pattern or not:

- e_1 is a pattern) Then we can use the (Bind) rule, obtaining $\mathcal{P} \vdash let\ X = e_1\ in\ e_2 \to^{lf} e_2[X/e_1]$.
- e_1 is not a pattern) Since $let\ X = e_1\ in\ e_2$ is ground we know that e_1 is ground (notice that this does not force e_2 to be ground). Moreover, $\mathcal{A} \vdash e_1 : \tau_t$, so by the Induction Hypothesis we can rewrite e_1 to some e_1': $\mathcal{P} \vdash e_1 \to^{lf} e_1'$. Using the (Contx) rule we can transform this local step into a step in the whole expression: $\mathcal{P} \vdash let\ X = e_1\ in\ e_2 \to^{lf} let\ X = e_1'\ in\ e_2$.

□

A.5 Proof of Theorem 3: Type Preservation

Proof. We proceed by case distinction over the rule of the *let*-rewriting relation $\to^{lf}$ (Figure 2) used to reduce e to e'.

(Fapp) If we reduce an expression e using the (Fapp) rule is because e has the form $f\ t_1\theta \ldots t_n\theta$ (being $f\ t_1 \ldots t_m \to r$ a rule in $\mathcal{P}$) and e' is $r\theta$. In this case we want to prove that $\mathcal{A} \vdash r\theta : \tau$. Since $wt_{\mathcal{A}}(\mathcal{P})$ then $wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e)$, and by the definition of well-typed rule (Definition 2) we have:

(A) $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t_1 \ldots t_m : \tau_L | \pi_L$

(B) $\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash r : \tau_R | \pi_R$

(C) $\exists \pi.\ (\tau_R, \overline{\beta_n \pi_R})\pi = (\tau_L, \overline{\alpha_n \pi_L})$

(D) $\mathcal{A}\pi_L = \mathcal{A}$, $\mathcal{A}\pi_R = \mathcal{A}$ and $\mathcal{A}\pi = \mathcal{A}$.

By the premises we have the derivation

$$(E)\mathcal{A} \vdash f\ t_1\theta \ldots t_m\theta : \tau$$

where $\theta = [\overline{X_n/t'_n}]$. Since the type derivation *(E)* exists, then there exists also a type derivation for each pattern t'_i: *(F)* $\mathcal{A} \vdash t'_i : \tau_i$.

If we replace every pattern t'_i in the type derivation *(E)* by their associated variable X_i and we add the assumptions $\{\overline{X_n : \tau_n}\}$ to $\mathcal{A}$, we obtain the type derivation:

$$(G)\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1 \ldots t_m : \tau$$

By *(A)* and *(G)* and Theorem 7 we have *(H)* $\exists\pi_1.\ (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_L\pi_1 = \mathcal{A} \oplus \{\overline{X_n : \tau_n}\}$ and $\tau_L\pi_1 = \tau$. Therefore $\mathcal{A}\pi_L\pi_1 = \mathcal{A}$ and $\alpha_i\pi_L\pi_1 = \tau_i$ for each i.

By *(B)* and the soundness of the inference (Theorem 6):

$$(I)\mathcal{A}\pi_R \oplus \{\overline{X_n : \beta_n\pi_R}\} \vdash r : \tau_R$$

Using the fact that type derivations are closed under substitutions (Theorem 8-a) we can add the substitution π of *(C)* to *(I)*, obtaining:

$$(J)\mathcal{A}\pi_R\pi \oplus \{\overline{X_n : \beta_n\pi_R\pi}\} \vdash r : \tau_R\pi$$

By *(J)* y *(C)* we have that *(K)* $\mathcal{A}\pi_R\pi \oplus \{\overline{X_n : \alpha_n\pi_L}\} \vdash r : \tau_L$

Using the closure under substitutions of type derivations (Theorem 8-a) we can add the substitution π_1 of *(H)* to *(K)*:

$$(L)\mathcal{A}\pi_R\pi\pi_1 \oplus \{\overline{X_n : \alpha_n\pi_L\pi_1}\} \vdash r : \tau_L\pi_1$$

By *(L)* and *(H)* we have *(M)* $\mathcal{A}\pi_R\pi\pi_1 \oplus \{\overline{X_n : \tau_n}\} \vdash r : \tau$

By $\mathcal{A}\pi_L = \mathcal{A}$ *(D)* and $\mathcal{A}\pi_L\pi_1 = \mathcal{A}$ *(H)* we know that *(N)* $\mathcal{A}\pi_1 = \mathcal{A}$.

From *(D)* and *(N)* follows *(O)* $\mathcal{A}\pi_R\pi\pi_1 = \mathcal{A}\pi\pi_1 = \mathcal{A}\pi_1 = \mathcal{A}$.

By *(O)* and *(M)* we have *(P)* $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash r : \tau$

Using Theorem 8-b) we can add the type assumptions $\{\overline{X_n : \tau_n}\}$ to the type derivations in *(F)*, obtaining *(Q)* $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t'_i : \tau_i$.

By Theorem 8-c) we can replace the data variables $\overline{X_n}$ in *(P)* by expressions of the same type. We use the patterns $\overline{t'_n}$ in *(Q)*:

$$(R)\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash r\theta : \tau$$

Finally, the data variables $\overline{X_n}$ do not appear in $r\theta$, so by Theorem 8-b) we can erase that assumptions in *(R)*:

$$(S)\mathcal{A} \vdash r\theta : \tau$$

(Ffail) and (FailP) Straightforward since in both cases e' is *fail*. A type derivation $\mathcal{A} \vdash \mathit{fail} : \tau$ is possible for any τ since $\mathcal{A}$ contains the assumption $\mathit{fail} : \forall\alpha.\alpha$.

The rest of the cases are the same as the proof in Enrique Martin-Martin's master thesis[6]

□

A.6 Lemma 2: Faulty Expressions are ill-typed

We use an auxiliary result stating that *junk* expressions cannot have a valid type wrt. any coherent set of assumptions $\mathcal{A}$.

Lemma 4. *If e is a* junk *expression* wrt. *a set of constructor symbols CS then there is no $\mathcal{A}$ coherent with CS such that $wt_{\mathcal{A}}(e)$.*

Proof. If e is *junk* then it has the form $c\ t_1 \ldots t_n$ with $c \in CS^m$ and $n > m$. Since application is left associative we can rewrite this expression as $(c\ t_1 \ldots t_m)\ t_{m+1} \ldots t_n$. In the type derivation of e appears a subderivation of the form:

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A} \vdash (c\ t_1 \ldots t_m) : \tau_1 \to \tau \\ \mathcal{A} \vdash t_{m+1} : \tau_1\end{array}}{\mathcal{A} \vdash (c\ t_1 \ldots t_m)\ t_{m+1} : \tau}$$

$\mathcal{A}$ is coherent with CS, so any possible type derived for the symbol c has the form $\tau'_1 \to \ldots \to \tau'_m \to (C\ \tau''_1 \ldots \tau''_k)$. Then after m applications of the **[APP]** rule the type derived for $c\ t_1 \ldots t_m$ is $C\ \tau''_1 \ldots \tau''_k$. This type is not a functional one as we expected $(\tau_1 \to \tau)$, so we have found a contradiction.

□

Proof of Lemma 2: Faulty Expressions are ill-typed

Proof. We prove it by contradiction. Suppose that e has a junk subexpression e' wrt. a set of constructor symbols CS coherent with $\mathcal{A}$ and we have a type derivation $\mathcal{A} \vdash e : \tau$. Therefore, in that derivation we have a subderivation $\mathcal{A}' \vdash e' : \tau'$ (for some $\mathcal{A}'$ and τ'). The set of assumptions $\mathcal{A}'$ is the result of adding some assumptions for variables to $\mathcal{A}$, so $\mathcal{A}'$ is also coherent with CS because the assumptions for data constructors have not change. By Lemma 4 those $\mathcal{A}'$ and τ' cannot exist, so we have found a contradiction.

□

[6] `http://gpd.sip.ucm.es/enrique/publications/master/masterThesis.pdf`

A.7 Theorem 4: Syntactic Soundness

We need some auxiliary results:

Lemma 5 (Well-typed normal forms are patterns). *If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$, e is ground and e is a normal form then e is a pattern.*

Proof. Straightforward from progress (Th. 2).

□

Lemma 6. *If $\mathcal{P} \vdash e \rightarrow^{lf} e'$ and $\mathcal{P}$ does not contains extra variables in its rules, then $fv(e') \subseteq fv(e)$.*

Proof. By case distinction over the rule applied in the step $\mathcal{P} \vdash e \rightarrow^{lf} e'$.

□

From the previous lemma follows an useful corollary:

Corollary 1. *If e is ground, $\mathcal{P} \vdash e \twoheadrightarrow^{lf} e'$ and $\mathcal{P}$ does not contains extra variables in its rules, then e' is ground.*

Proof of Theorem 4: Syntactic Soundness

Proof. Let e' be an arbitrary expression in $nf_{\mathcal{P}}(e)$. Since e is ground, we know by Corollary 1 that e' is ground. By Type Preservation (Th. 3) we also know that $\mathcal{A} \vdash e' : \tau$. Therefore by Lemma 5 we know that e' must be a pattern.

□

A.8 Proof of Theorem 5

Proof. By contradiction. Suppose that $\mathcal{A} \vdash e : \tau$, $wt_{\mathcal{A}}(\mathcal{P})$, e is ground and there exists some e' such that $\mathcal{P} \vdash e \twoheadrightarrow^{lf} e'$ and e' is faulty. By Type Preservation (Th. 3) we know that $\mathcal{A} \vdash e' : \tau$, but by Lemma 2 faulty expressions are ill-typed, reaching a contradiction.

□