Under consideration for publication in Math. Struct. in Comp. Science

A Liberal Type System for Functional Logic Programs[†]

FRANCISCO JAVIER LÓPEZ-FRAGUAS[1],

ENRIQUE MARTIN-MARTIN[1] *and*

JUAN RODRÍGUEZ-HORTALÁ[1]

[1] *Dpto. de Sistemas Informáticos y Computación, Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain.*

Received 17 July 2012

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, a particular form of 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.

1. Introduction

Functional logic programming. Functional logic languages (Hanus, 2007) like Toy (López-Fraguas and Sánchez-Hernández, 1999) or Curry (Hanus (ed.), 2006) have a strong resemblance to lazy functional languages like Haskell (Hudak et al., 2007). 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 (González-Moreno et al., 1999) 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*:

$$f\ X \to X \qquad f\ X \to s\ X \qquad double\ X \to add\ X\ X$$

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

[†] This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502)

choice means that all copies of a non-deterministic subexpression ($f\ z$ in the example) created during reduction share the same value.

In the HO-CRWL[†] approach to FLP (González-Moreno et al., 1997), followed by the Toy system, programs can use *HO-patterns* (essentially, partial applications of function or constructor symbols to other patterns) in left hand sides of function definitions. These patterns are treated in a purely syntactic way, so problems of HO unification are avoided. HO patterns correspond 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 (López-Fraguas et al., 2008) 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: (González-Moreno et al., 2001) and (López-Fraguas et al., 2010) considered that problem, and this paper makes also some contributions in this sense.

All those aspects of FLP play a role in the paper, and Section 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. For example, our operational semantics (Section 3.1) supports evaluation of open expressions, i.e., expressions containing free variables, which are forbidden in functional programming. However this feature does not play any relevant role in this paper, so readers can assume that all expressions to reduce are closed.

Types, FLP and genericity. FLP languages are typed languages adopting classical Damas-Milner types (Damas and Milner, 1982). 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 ... (Hudak et al., 2007) Some exceptions to this fact are some preliminary proposals for type classes in FLP (Moreno-Navarro et al., 1996; Lux, 2008), where in particular a technical treatment of the type system is absent.

By the term *generic programming* 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 the 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 (Wadler and Blott, 1989) were invented to deal with those situations. Some further developments of the idea of generic programming (Hinze, 2006) are based on type classes, while others (Hinze and Löh, 2007) have preferred to use simpler extensions of Damas-Milner system, such as GADTs (Cheney and Hinze, 2003; Schrijvers et al., 2009). 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 num-

[†] CRWL (González-Moreno et al., 1999) stands for *Constructor Based Rewriting Logic*; HO-CRWL is a higher order extension of it.

ber of constructor occurrences in its argument is an easy task in a type-free language with functional syntax:

$$\begin{array}{ll} \mathit{size\ true} \to \mathit{s\ z} & \mathit{size\ false} \to \mathit{s\ z} \\ \mathit{size\ z} \to \mathit{s\ z} & \mathit{size\ (s\ X)} \to \mathit{s\ (size\ X)} \\ \mathit{size\ [\,]} \to \mathit{s\ z} & \mathit{size\ (X{:}Xs)} \to \mathit{s\ (add\ (size\ X)\ (size\ Xs))} \end{array}$$

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 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 (Hinze and Löh, 2007). This kind of encoding is also supported by our system (see the *show* function in Example 3.1 and *eq* in Figure 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 $\forall\alpha.\alpha \to \mathit{nat}$, of which each rule of *size* gives a more concrete instance. A detailed discussion of the advantages and disadvantages of such liberal declarations appears in Sections 4 and 6.

The proposed well-typedness criterion for programs proceeds rule by rule and requires only a quite simple additional check over usual Damas-Milner type inference performed over both sides of each rule. Here, 'simple' does not mean 'naive'. For example, 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. To illustrate this, consider the rule $f\ X \to \mathit{not}\ X$ with the assumptions $f : \forall\alpha.\alpha \to \mathit{bool}$, $\mathit{not} : \mathit{bool} \to \mathit{bool}$. The type of the rule is $\mathit{bool} \to \mathit{bool}$, 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 $\mathit{not}\ z$. 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. Definition 3.1 in Section 3.3 states that point with precision, and allows us to prove type soundness for our system. As we will also see in Section 4, our conditions are in some technical sense the most liberal suitable conditions under which reduction preserve types.

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 Section 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 a particular form of existential types and GADT-like encodings, not available in current FLP systems. Moreover, the use of HO-patterns is ensured to be type-safe, while in current FLP systems it is either unrestricted (and therefore unsafe) or forbidden because of those type-safety problems.

— Section 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, and we give a result of maximal liberality while keeping type preservation; then by introducing

failure rules to the formal operational calculus, we are also able to ensure the *progress* property of well-typed expressions. Based on those results we also state *syntactic soundness* of the type system, in the sense of (Wright and Felleisen, 1992).

— In Section 5 we give a significant collection of examples showing the interest of the proposal. These examples cover type-indexed functions (with an application to the implementation of type classes), existential types, opaque higher-order patterns and generic functions. None of them is supported by existing FLP systems.

— The well-typedness criterion given in this paper provides a valuable alternative to (López-Fraguas et al., 2010) in the management of type-unsoundness problems due to the use of *HO-patterns* in function definitions. Both works, which are technically compared at the end of Section 3.3, improve largely the solutions given previously in (González-Moreno et al., 2001). As concrete advantages of the proposal in this paper, we can type equality, solving known problems of *opaque decomposition* (González-Moreno et al., 2001) (Section 5.1) and, most remarkably, we can type the *apply* function appearing in the HO-to-FO translation used in standard FLP implementations (Section 5.2).

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

This is a revised and extended version of a previous conference paper (López-Fraguas et al., 2010).

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 also proposed for GADTs (Cheney and Hinze, 2003; Peyton Jones et al., 2006). We also assume a denumerable set $\mathcal{DV}$ of *data variables* X. The notation $\overline{o_n}$ stands for a sequence of n objects $o_1, \ldots, o_n$, where o_i is the i^{th} element in the sequence. Figure 1 shows the syntax of *patterns* $\in Pat$—our notion of values—and *expressions* $\in Exp$. The role of let-bindings is to express sharing of subexpressions, as corresponds to call-time choice semantics. We split the set of patterns in two: *first order patterns* $FOPat \ni fot ::= X \mid c\ fot_1 \ldots fot_n$ 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 $c\ e_1 \ldots e_n$ are called *junk* if $n > ar(c)$ and $c \neq fail$, and expressions $f\ e_1 \ldots e_n$ are called *active* if $n \geq ar(f)$. The set $fv(e)$ of *free variables* of an expression e is defined in the usual way as the set of variables in e which are not bound by any let construction; notice that free variables in let-bindings are defined as $fv(let\ X = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus \{X\})$, corresponding to the fact that we do not consider recursive let-bindings. 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

Data variables			$X, Y, Z, \ldots$
Type variables			$\alpha, \beta, \gamma, \ldots$
Data constructors			c
Type constructors			C
Function symbols			f
Symbol	s	::=	$X \mid c \mid f$
Non variable symbol	h	::=	$c \mid f$
Expressions	e	::=	$X \mid c \mid f \mid e\ e$
		$\mid$	$let\ X = e\ in\ e$
Patterns	t	::=	X
		$\mid$	$c\ t_1 \ldots t_n$ if $n \leq ar(c)$
		$\mid$	$f\ t_1\ \ldots t_n$ if $n < ar(f)$
Data substitution	θ	::=	$[\overline{X_n/t_n}]$
Program rule	R	::=	$f\ \bar{t} \to e$ $\ (\bar{t}$ linear$)$
Program	$\mathcal{P}$	::=	$\{R_1, \ldots, R_n\}$
Simple Types	τ	::=	α
		$\mid$	$C\ \tau_1 \ldots \tau_n$ if $ar(C) = n$
		$\mid$	$\tau_1 \to \tau_2$
Type Schemes	σ	::=	$\forall\overline{\alpha_n}.\tau$
Type substitution	π	::=	$[\overline{\alpha_n/\tau_n}]$
Assumptions	$\mathcal{A}$	::=	$\{s_1 : \sigma_1, \ldots, s_n : \sigma_n\}$

Fig. 1. Syntax of expressions, programs and types.

substitution is written as id. A *program rule* R is defined as $f\ \overline{t_n} \to e$ (we also refer to rules as $f\ \overline{t_n} \to r$ or $l \to r$) 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. Since the constructor *fail* is an artifact conceived to deal properly with *progress* properties of the type system in Section 4, *fail* is not supposed to occur in program rules, although it would not produce any technical problem. A program $\mathcal{P}$ is a set of program rules: $\{R_1, \ldots, R_n\} (n \geq 0)$.

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 $ar(C) = n$. Figure 1 shows the syntax of *simple types* τ and *type-schemes* σ. The set of *free type variables (ftv)* of a simple type τ is $var(\tau)$, and for type-schemes $ftv(\forall\overline{\alpha_n}.\tau) = ftv(\tau) \setminus \{\overline{\alpha_n}\}$. A type-scheme σ is *closed* if $ftv(\sigma) = \emptyset$. A *set of assumptions* $\mathcal{A}$ is $\{\overline{s_n : \sigma_n}\}$ fulfilling that $\mathcal{A}(fail) = \forall\alpha.\alpha$ and for every c in $CS^n \setminus \{fail\}$, $\mathcal{A}(c) = \forall\overline{\alpha}.\tau_1 \to \ldots \to \tau_n \to (C\ \tau'_1 \ldots \tau'_m)$ for some type constructor C with $ar(C) = m$. Therefore the type assumptions for constructors must correspond to their arity and, as in (Cheney and Hinze, 2003; Peyton Jones et al., 2006), the constructor *fail* can have any type. $\mathcal{A}(s)$ denotes the type-scheme associated to symbol s, and 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

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

(Ffail) $f\ t_1 \ldots t_n \rightarrowtail \mathit{fail}$, 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 $f\ t_1 \ldots t_n$ are unifiable

(FailP) $\mathit{fail}\ e \rightarrowtail \mathit{fail}$

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

(Bind) $\mathit{let}\ X = t\ \mathit{in}\ e \rightarrowtail e[X/t]$

(Elim) $\mathit{let}\ X = e_1\ \mathit{in}\ e_2 \rightarrowtail e_2$, if $X \notin fv(e_2)$

(Flat) $\mathit{let}\ X = (\mathit{let}\ Y = e_1\ \mathit{in}\ e_2)\ \mathit{in}\ e_3 \rightarrowtail \mathit{let}\ Y = e_1\ \mathit{in}\ (\mathit{let}\ X = e_2\ \mathit{in}\ e_3)$, if $Y \notin fv(e_3)$

(LetAp) $(\mathit{let}\ X = e_1\ \mathit{in}\ e_2)\ e_3 \rightarrowtail \mathit{let}\ X = e_1\ \mathit{in}\ e_2\ e_3$, if $X \notin fv(e_3)$

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

Fig. 2. **Higher order *let*-rewriting relation with pattern matching failure $\rightarrowtail$**

symbols not appearing in $\mathcal{A}'$ (notice that $\oplus$ is not commutative). For sets of assumptions, free type variables are defined as $ftv(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. Notice that type-schemes for data constructors may be existential, i.e., they can be of the form $\forall\overline{\alpha_n}.\overline{\tau_m} \to \tau$ where $(\bigcup_{i=1}^{m} ftv(\tau_i)) \setminus ftv(\tau) \neq \emptyset$. 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 that 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. *Operational semantics*

The operational semantics of our programs is based on *let*-rewriting (López-Fraguas et al., 2008), a high level notion of reduction step devised to express call-time choice through the use of let-bindings that represent subexpression sharing. For this paper, we have extended *let*-rewriting with two rules for managing failure of pattern matching (Figure 2), playing a role similar to the rules for pattern matching failure in GADTs (Cheney and Hinze, 2003; Peyton Jones et al., 2006). We write $\rightarrowtail$ for the extended relation and $\mathcal{P} \vdash e \rightarrowtail e'$ ($\mathcal{P} \vdash e \rightarrowtail^* e'$ resp.) to express one step (zero or more steps resp.) of $\rightarrowtail$ using the program $\mathcal{P}$. By $nf_{\mathcal{P}}(e)$ we denote the set of *normal forms* reachable from e, i.e., $nf_{\mathcal{P}}(e) = \{e' \mid \mathcal{P} \vdash e \rightarrowtail^* e' \text{and } e' \text{ is not } \rightarrowtail\text{-reducible}\}$. Notice that *let*-rewriting can reduce expressions with free variables (open expressions), although it does not bind them to values. However this support for open expressions does not play any relevant role in this paper, which can be understood as if all expressions to reduce were closed.

The new rule (Ffail) generates a failure when no program rule can be used to reduce

a function application. Notice the use of syntactic 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). To see that, consider for instance the program

$$true \wedge X \to X \qquad false \wedge X \to false$$

and the expression $let\ Y = true\ in\ (Y \wedge true)$. The subexpression $Y \wedge true$ unifies with the function rule left-hand side $true \wedge X$, so no failure is generated. If we use pattern matching as condition without considering the binding $Y = true$, a failure is incorrectly generated since none of the left-hand sides $true \wedge X$ and $false \wedge X$ matches the subexpression $Y \wedge true$. Besides, using unification in (Ffail) also contributes to early detection of proper failures. Consider the program $\mathcal{P}_2 = \{f\ true\ false \to true, loop \to loop\}$ and the expression $let\ Y = loop\ in\ f\ Y\ Y$. Since $f\ Y\ Y$ does not unify with $f\ true\ false$, (Ffail) detects a failure, while other operational approaches to failure in FLP (Sánchez-Hernández, 2006) would lead to divergence.

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

Extending the *let*-rewriting relation of (López-Fraguas et al., 2008) 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 (x:xs)* $\to x$, where the case *head [ ]* is (intentionally) missing. Similar to what happens in FP systems like Haskell, we expect *(head [ ])* to give raise to a failing reduction, but not to a type error. A difference is that in FP an attempt to evaluate *(head [ ])* 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 Section 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.

[‡] As mentioned in Section 1, patterns in our setting (both first and higher order patterns) are treated in a purely syntactic way, so syntactic unification is used instead of more complex HO unification procedures.

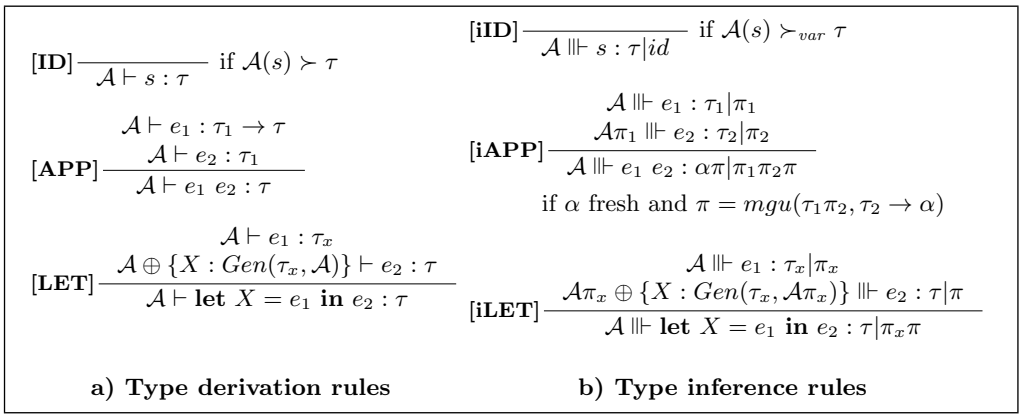

Fig. 3. Type system

3.2. *Type derivation and inference for expressions*

Both derivation and inference rules are based on those presented in (López-Fraguas et al., 2010). Our type derivation rules for expressions (Figure 3-a) correspond to the well-known variation of Damas-Milner's (Damas and Milner, 1982) type system with syntax-directed rules, so there is nothing essentially new here—the novelty will come from the notion of well-typed program given in Definition 3.1 below. $Gen(\tau, \mathcal{A})$ 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 $\Vdash$ (Figure 3-b) follows the same ideas as the algorithm $\mathcal{W}$ (Damas and Milner, 1982). We have given a relational style to type inference 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 as inputs 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. This definition formalizes in terms of type derivations and substitutions the intuitive well-typedness idea explained in Section 1: right-hand sides of program rules must not restrict the types of variables more than left-hand sides.

Definition 3.1 (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 there exist π_L, τ_L, π_R and τ_R such that:

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 independently have valid types by assigning *some* types to variables, obtaining the most general ones for them in both sides, but not imposing any relationship between them. This is left to the third point, which is the most important one. It checks that the obtained most general types for the right-hand side and the variables appearing in it are more general than the obtained ones for the left-hand side. This point, which avoids that right-hand sides restrict the types of variables more than left-hand sides, guarantees the *type preservation* property (i.e., that 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 *opaque variables* (López-Fraguas et al., 2010)—either introduced by the presence of existentially quantified constructors or HO-patterns—which results in the support of a particular variant of existential types (Läufer and Odersky, 1994)—see Section 5.2 for more details. Finally, the last point guarantees that free variables in the set of assumptions are not modified by neither the most general typing substitutions of both sides nor the matching substitution. In practice, this point holds trivially if type assumptions for program functions are closed, as it is usual. Points i) and ii) in the previous definition have are very declarative formulation, but are not particularly well suited to the effective implementation of the well-typedness check. Thanks to the close relationship between type derivation and inference for expressions—soundness and completeness, Theorems A.1 and A.2 in page 26—we can recast points i) and ii) of Definition 3.1 in a more operational and oriented to implementation style.

Definition 3.2 (Well-typed program wrt. $\mathcal{A}$; alternative formulation).
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 there exist π_L, τ_L, π_R and τ_R such that:

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.

Now, 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; so the

implementation is straightforward. The equivalence between both definitions of well-typed rule follows easily from the following result about type derivation and inference:

Lemma 3.1. 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 ($\mathcal{A}\pi \vdash e : \tau$) $\Longleftrightarrow \exists \pi', \tau'$ such that $\mathcal{A} \Vdash e : \tau' | \pi'$, where π, π' (τ, τ' respectively) are equal up to variable renaming.

Proof. Straightforward based on soundness and completeness of the inference relation wrt. to type derivation (Theorem A.1 and Theorem A.2 in Appendix A). □

Both definitions of well-typed rule present some similarities with the notion of *typeable rewrite rule* for Curryfied Term Rewriting Systems in (van Bakel and Fernández, 1997). 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. This condition is similar to points 1–3 of our definition, which force the most general types obtained for the right-hand side to be more general than those inferred for the right-hand side. However, Definition 3.2 provides a more effective procedure to check well-typedness than the notion of typeable rewrite rule. On the other hand (van Bakel and Fernández, 1997) considers a different setting that includes intersection types, not addressed in our work.

Example 3.1 (Well and ill-typed rules and expressions). Let us consider the following assumptions and program:

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

$$\begin{array}{l}\mathcal{P} \equiv \{\ id\ X \to X,\ snd\ X\ Y \to Y,\ unpack\ (snd\ X) \to X,\ eq\ (s\ X)\ z \to false,\\ show\ rnat\ X \to showNat\ X,\ f\ true \to z,\ f\ true \to false,\\ flist\ [z] \to s\ z,\ flist\ [true] \to false\ \}\end{array}$$

It is easy to see that the rules for the functions *id* and *snd* are well-typed. The function *unpack* is taken from (González-Moreno et al., 2001) as a typical example of the type problems that HO-patterns can produce. According to Definition 3.2 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 $unpack\ (snd\ true) \rightarrowtail true$, 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 $f\ true \rightarrowtail z$; $f\ true$ can have any type, in particular $bool$, but z can only have type nat. Both rules of function $flist$ are well-typed, however its type assumption cannot be made more general for its first argument: it can be seen that there is no τ such that 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 many such expressions, that typically correspond to programming errors. 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.

In (López-Fraguas et al., 2010) we extended Damas-Milner types with some extra control over HO-patterns, leading to another definition of well-typed programs, written $wt^{old}_{\mathcal{A}}(\mathcal{P})$ here. All valid programs in (López-Fraguas et al., 2010) are still valid:

Theorem 3.1. If $wt^{old}_{\mathcal{A}}(\mathcal{P})$ then $wt_{\mathcal{A}}(\mathcal{P})$.

Proof. See page 27 in Appendix A. □

To further appreciate the difference between the two approaches, notice that all the examples in Section 5 are rejected as ill-typed by (López-Fraguas et al., 2010). The purpose of the two systems is different: in this paper we attempt deliberately to go beyond Damas-Milner, while (López-Fraguas et al., 2010) only aims to deal safely with programs using HO-patterns in rules, but keeping the behavior of Damas-Milner otherwise. In correspondence to that, in (López-Fraguas et al., 2010) the types of program functions can be inferred, while in the present work they must be explicitly declared.

4. Properties of the type system

We will follow two alternative approaches for proving type soundness of our system. First, we prove the theorems of *progress* and *type preservation* similar to those that play the main role in the type soundness proof for GADTs (Cheney and Hinze, 2003; Peyton Jones et al., 2006). After that, we follow a syntactic approach similar to (Wright and Felleisen, 1992). The first result, *progress*, states that well-typed ground expressions are either patterns or expressions reducible by *let*-rewriting.

Theorem 4.1 (Progress). If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$ and e is ground, then either e is a pattern or $\exists e'.\ \mathcal{P} \vdash e \rightarrowtail e'$.

Proof. By induction over the structure of e, see page 29 in Appendix A for the complete proof. □

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 4.2 (Type Preservation). If $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ and $\mathcal{P} \vdash e \rightarrowtail e'$, then $\mathcal{A} \vdash e' : \tau$.

Proof. By case distinction over the rule of the let-rewriting relation $\rightarrowtail$ used to reduce e to e'. The detailed proof can be found in page 31 in Appendix A. □

This result shows that the degree of liberality given to our type system is not arbitrary: types are certainly more liberal than in the usual Damas-Milner system, but they are also restricted enough as to ensure that types are not lost during reduction. In Example 3.1 we saw examples of ill-typed programs for which type preservation fails. At this point, an interesting question arises: could the type system be even more relaxed but still keep type preservation? The following results shows that in a certain sense the answer is *'no'*, and therefore our well-typedness conditions are as liberal as possible without compromising type preservation.

Theorem 4.3 (Maximal liberality of well-typedness conditions).
Let $\mathcal{A}$ be a closed set of assumptions, and assume that $\mathcal{P}$ is a program which is not well-typed wrt. $\mathcal{A}$, but such that every rule $R \in \mathcal{P}$ verifies the condition i) of well-typedness in Definition 3.2. Then there exists a rule $(f\ t_1 \ldots t_m \rightarrow e) \in \mathcal{P}$ with variables $\overline{X_n}$ and there exist types $\overline{\tau_n}$, τ such that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1 \ldots t_m : \tau$ and $f\ t_1 \ldots t_m \rightarrowtail e$ but $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \nvdash e : \tau$.

Proof. By case distinction on the condition of $wt_{\mathcal{A}}(\mathcal{P})$ that fails. The complete proof can be found in page 32 in Appendix A. □

By requiring the condition that all rules in the program verify condition i) of program well-typedness, we ensure that ill-typedness of the program is not due to a badly typed left-hand side of a rule—an uninteresting case from the point of view of type preservation under reduction—but must be due to a failure of conditions ii) or iii)—as condition iv) does not fail for closed assumptions—that is, due to a lack of right correspondence between some left-hand side and its companion right-hand side. We remark that the proof of Theorem 4.3 is constructive in the sense that, for a program in the hypothesis of the theorem, it provides explicitly a reduction step and types which witness the failure of type preservation.

Theorem 4.3 also indicates that, in a sense, our notion of well-typed rule captures essentially the intuitive idea that a rule preserves types when applied to reduce an expression. That intuition becomes indeed a provable technical result by giving a declarative definition of type-preserving rule and proving that, under certain reasonable conditions, this notion is equivalent to well-typedness.

Definition 4.1 (Type-preserving rule). Given a set of assumptions $\mathcal{A}$, we say that a rule $f\ t_1 \ldots t_m \rightarrow e$ *preserves types* if

(i) its left-hand side admits some type, i.e., $wt_{\mathcal{A}\oplus\{\overline{X_n:\tau_n}\}}(f\ t_1 \ldots t_m)$ for some $\overline{\tau_n}$, where $\overline{X_n}$ are the variables appearing in the rule—$\{\overline{X_n}\} = fv(f\ t_1 \ldots t_m)$.
(ii) $\mathcal{A} \vdash f\ t_1\theta \ldots t_m\theta : \tau \Longrightarrow \mathcal{A} \vdash e\theta : \tau$, for any substitution θ and type τ.

We impose the first condition to avoid the case of rules which do not break type preservation trivially because their left-hand sides are not well-typed, so that $\mathcal{A} \not\vdash f\ t_1\theta \ldots t_m\theta : \tau$ for any τ.

The notions of well-typed rules and type-preserving rules are equivalent, but only for a certain kind of assumptions which are rich enough to build monomorphic terms of any given type, as formalized in the following definition.

Definition 4.2 (Type-complete set of assumptions). A set of assumptions $\mathcal{A}$ is called *type-complete* if for each simple type τ there exists a pattern t_τ which can only have that type, i.e., $\mathcal{A} \vdash t_\tau : \tau$ and $\mathcal{A} \not\vdash t_\tau : \tau'$ for all $\tau' \neq \tau$.

Now, we can prove the announced equivalence result, showing that the definition of well-typed rule capture algorithmically the precise declarative notion of type preservation in function applications.

Proposition 4.1. Consider a type-complete set of assumptions $\mathcal{A}$, and a program rule R. Then R preserves types iff $wt_{\mathcal{A}}(R)$.

The condition of type-completeness is imposed to avoid cases when type preservation in a function application is potentially compromised but not actually broken *with the data constructors and functions currently in the program.* However, if the program is extended with new symbols, it would be possible to call the function breaking type preservation. The following example shows this situation:

Example 4.1. Consider the program $\mathcal{P} \equiv \{id\ X \to X, f\ F \to F\ true\}$ with types $\mathcal{A} \equiv \{id : \forall\alpha.\alpha \to \alpha, f : \forall\alpha.(\alpha \to \alpha) \to bool\}$. It is easy to check that, with the current data constructor and functions symbols, the only pattern that can be passed as argument of f making the application well-typed is id, which preserves types. However types are not preserved for any pattern whose only type was $\tau \to \tau$ (for any τ). If we add to the program the function $\{inc\ N \to N + 1\}$ with type $int \to int$ then the rule for f break type preservation: $\mathcal{A} \vdash f\ inc : bool$ but $\mathcal{A} \not\vdash inc\ true : bool$.

Notice that according to the definition of well-typed rule (Definitions 3.1 or 3.2) the rule for f is ill-typed in both situations, as the right-hand side restricts the type of F more than its left-hand side—although in the first case there is not enough symbols to cause the loss of type preservation.

We now turn to a syntactic approach to type safety similar to (Wright and Felleisen, 1992). Before that we need to define some properties about expressions:

Definition 4.3. 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*. Not all faulty expressions are stuck. For example, *snd* $(z\ z)$ *true* is *faulty* but *snd* $(z\ z)$ *true* $\rightarrowtail$ *true*. However all faulty expressions are ill-typed:

Lemma 4.1 (Faulty expressions are ill-typed). If e is faulty then there is no $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$.

Proof. By contradiction, using the fact that *junk* expressions cannot have a valid type wrt. any set of assumptions $\mathcal{A}$. See page 34 in Appendix A for a complete proof. □

The next theorem states that all finished reductions of well-typed ground expressions do not get stuck but end up in patterns of the same type as the original expression.

Theorem 4.4 (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 $\mathcal{A} \vdash e' : \tau$.

Proof. See page 35 in Appendix A for a complete proof. □

The following complementary result states that the evaluation of well-typed expressions does not pass through any faulty expression.

Theorem 4.5. If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$ and e is ground, then there is no e' such that $\mathcal{P} \vdash e \rightarrowtail^* e'$ and e' is faulty.

Proof. By contradiction. Suppose that $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$, e is ground and there exists some e' such that $\mathcal{P} \vdash e \rightarrowtail^* e'$ and e' is faulty. By Type Preservation (Theorem 4.2) we know that $\mathcal{A} \vdash e' : \tau$, but by Lemma 4.1 faulty expressions are ill-typed, reaching a contradiction. □

4.1. *Discussion of the properties*

We discuss now the strength of our results considering some interdependent factors: the rules for failure in Section 3, the liberality of our well-typedness condition, and our notion of faulty expression.

Progress and type preservation. In (Milner, 1978) Milner considered *'a value 'wrong', which corresponds to the detection of a failure at run-time'* to reach his famous lemma *'well-typed programs don't go wrong'*. For this to be true in languages with pattern matching, like Haskell or ours, not all run-time failures should be seen as wrong, as happens with definitions like *head (x:xs)* $\rightarrow$ *x*, where there is no rule for *(head [ ])*. Otherwise, progress does not hold and some well-typed expressions become stuck. A solution is considering a 'well-typed completion' of the program, adding a rule like *head [ ]* $\rightarrow$ *error* where *error* is a value accepting any type. With it, *(head [ ])* reduces to *error* and is not wrong, but *(head true)*, which is ill-typed, is wrong and its reduction gets stuck. In our setting, completing definitions would be more complex because of HO-patterns that could lead to an infinite number of 'missing' cases. To cope with this problem, our failure rules in Section 3 are used to replace the 'well-typed completion'. We prefer the word *fail* instead of *error* because, in contrast to FP systems where an attempt to evaluate *(head [ ])* results in a run-time error, in FLP systems rather than an error this is a silent

failure in a possible space of non-deterministic computations managed by backtracking. Admittedly, in our system the difference between 'wrong' and 'fail' is weaker from the point of view of reduction. Certainly, junk expressions are stuck but, for instance, *(head [ ])* and *(head true)* both reduce to *fail*, instead of the ill-typed *(head true)* getting stuck. 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*. This cannot happen, however, because *type preservation* holds step-by-step, and then no reduction $e \to^* \mathit{fail}$ starting with a well-typed e can pass through the ill-typed *(head true)* as intermediate (sub)-expression.

Liberality. In our system the risk of accepting as well-typed some expressions that one might prefer to reject at compile time is higher than in more restrictive type systems. Consider the function *size* of Section 1, page 3. For any well-typed expression e, *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 is a pattern matching failure similar to the case of *(head [ ])*. This can be appreciated as a weakness, and is further discussed in Section 6 in connection to type classes and GADTs.

Syntactic soundness and faulty expressions. Theorems 4.4 and 4.5 are easy consequences of progress and type preservation. Theorem 4.5 is indeed a weaker safety criterion, because our faulty expressions only capture the presence of junk, which by no means is the only source of ill-typedness. For instance, the expressions *(head true)* or *(eq true z)* are ill-typed but not faulty. Theorem 4.5 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. Still, Theorem 4.5 contains no trivial information. Although checking the presence of junk is trivial (counting arguments suffices for it), the fact that a given expression will not become faulty during reduction is a typically undecidable property approximated by our type system. For example, consider g with type $\forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta$, defined as $g\ H\ X \to H\ X$. The expression *(g true false)* is not faulty but reduces to the faulty *(true false)*. Our type system avoids that because the non-faulty expression *(g true false)* is detected as ill-typed.

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 $\mathcal{P}$. We consider the following initial set of assumptions, common to all examples:

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

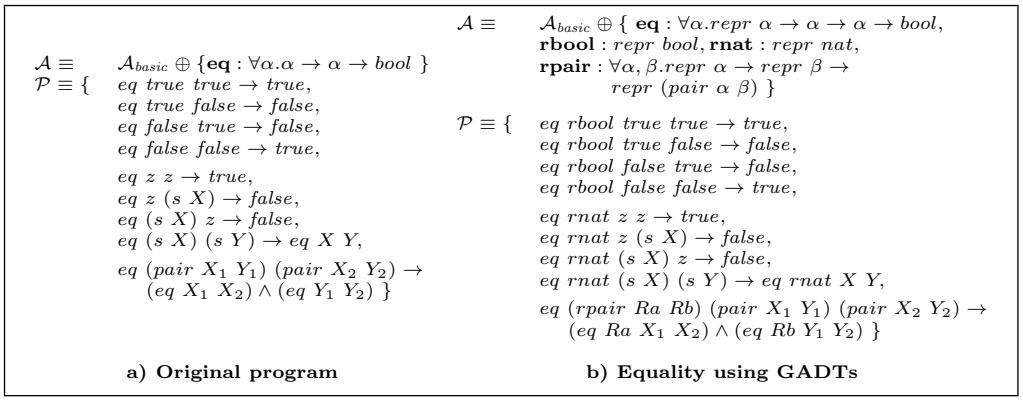

Fig. 4. Type-indexed equality

5.1. *Type-indexed functions*

Type-indexed functions—in the sense appeared in (Hinze and Löh, 2007)—are functions that have a particular definition for each type in a certain family. The function *size* of Section 1—page 3—is an example of such a function. A similar example is given in Figure 4-a, containing the code for an equality function which operates only 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 (Cheney and Hinze, 2003; Hinze and Löh, 2007). 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 those inferred 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. (Hanus, 2007)—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 Figure 4-a would be rejected by systems supporting GADTs (Cheney and Hinze, 2003; Schrijvers et al., 2009), while the encoding of equality using GADTs as type representations in Figure 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 Figure 4-a. With our proposed type system programmers can define structural equality as in Figure 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 (González-Moreno et al., 2001) that in the presence of HO-patterns[§] 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 (González-Moreno et al., 2001), but all of them need fully type-annotated expressions at run time, which penalizes efficiency. With our 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 $(bool, \alpha, \beta)$.

Finally, type-indexed functions in our type system have a very interesting application. It is well known that type classes (Wadler and Blott, 1989; Hall et al., 1996) provide a clean, modular and elegant way of writing overloaded functions in functional languages as Haskell. Type classes are usually implemented by means of a source-to-source transformation that introduces extra parameters—called dictionaries—to overloaded functions (Wadler and Blott, 1989; Hall et al., 1996). However, this classical translation produces a problem of missing answers when applied to FLP due to a bad interaction between non-determinism and the call-time choice semantics (Lux, 2009; Martin-Martin, 2011). Using type-indexed functions and *type witnesses*—a representation of types as values—it is possible to develop a type-passing translation for type classes similar to (Thatté, 1994) that solves this problem and whose translated programs are well-typed in the proposed liberal type system. Figure 5 shows the translation of a program with type classes using the equality class and function. As can be seen, the `eq` function is translated into a type-indexed function whose first argument is a type witness. These type witnesses—which are new constructors generated for the data types in program, with types `#bool`:: *bool* and `#list`:: $A \to [A]$—are used to determine which rules of the type-indexed function `eq` can be used. Proper type witnesses are passed to overloaded functions, as in the case of the `member` function. These witnesses are determined by a type analysis over the expressions in source programs, just as it is done in the classical dictionary-based translation of type classes.

Apart from solving the problem of missing answers, this type-passing translation also produces faster and simpler programs than the classical translation. A complete discussion of these points, the formalization of the translation and further examples can be found in (Martin-Martin, 2011).

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

```
eqBool :: bool → bool → bool
eqBool true true = true
eqBool true false = false
eqBool false true = false
eqBool false false = true

class eq A where
  eq :: A → A → bool

instance eq bool where
  eq X Y = eqBool X Y

instance ⟨eq A⟩ ⇒ eq [A] where
  eq [] [] = true
  eq [] (Y:Ys) = false
  eq (X:Xs) [] = false
  eq (X:Xs) (Y:Ys) =
    and (eq X Y) (eq Xs Ys)

member :: ⟨eq A⟩ ⇒ [A] → A → bool
member [] Y = false
member (X:Xs) Y =
  or (eq X Y) (member Xs Y)
```

a) Source program

```
eqBool :: bool → bool → bool
eqBool true true = true
eqBool true false = false
eqBool false true = false
eqBool false false = true

eq :: A → A → A → bool
eq #bool X Y = eqBool X Y
eq (#list W_A) [] [] = true
eq (#list W_A) [] (Y:Ys) = false
eq (#list W_A) (X:Xs) [] = false
eq (#list W_A) (X:Xs) (Y:Ys) = and
  (eq W_A X Y)
  (eq (#list W_A) Xs Ys)

member :: A → [A] → A → bool
member W_A [] Y = false
member W_A (X:Xs) Y =
  or (eq W_A X Y) (member W_A Xs Y)
```

b) Translated program

Fig. 5. Translation of a program using equality

5.2. *Existential types, opacity and HO-patterns*

Existential types (Mitchell and Plotkin, 1988; Perry, 1991; Läufer and Odersky, 1994) 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, i.e., it has the equivalent type $(\exists\alpha.\alpha \to (\alpha \to nat)) \to key$. This type means that the first argument of key is an expression of some *unknown* type α, and the second one is a function from that unknown type to natural numbers $(\alpha \to nat)$. Systems supporting existential types treat differently constructors with existential type (in the sequel *existential constructors*) depending on their place in the rule. If they appear in the right-hand side, they are treated as any other polymorphic symbol, allowing any instance of their type. However, if they appear in the left-hand side, new distinct constant types—called *Skolem constants*—are introduced for each existentially quantified variable. For example in $key\ X\ F$ the constructor key is assigned the type $\kappa \to (\kappa \to nat) \to key$—where κ is a fresh Skolem constant—so X and F have types κ and $\kappa \to nat$ respectively. Therefore, any occurrence of these data variables in the right-hand side that needs a more concrete type as $(not\ X)$ or $(F\ true)$ will be considered ill-typed. This situation also happens in the left-hand side of the rule, if key contains arguments of more concrete types as in $(key\ z\ s)$.

The type system presented in this paper accepts classical functions dealing with exis-

tential constructors, like *getKey*:

$$\mathcal{A} \equiv \mathcal{A}_{basic} \oplus \{\ \mathbf{getKey} : key \to nat\ \} \qquad \mathcal{P} \equiv \{\ getKey\ (key\ X\ F) \to F\ X\ \}$$

Notice that this rule is well-typed because the right-hand side does not force the types of the variables X and F (α and $\alpha \to \beta$ resp.) more than the left-hand side does (α and $\alpha \to nat$ resp.). However, the type system presented here gives a more permissive treatment to existential constructors than usual approaches (Mitchell and Plotkin, 1988; Perry, 1991; Läufer and Odersky, 1994). As a consequence, rules containing existential constructors with arguments of concrete types—as $getKey\ (key\ z\ s) \to z$ or $getKey\ (key\ (s\ X)\ F) \to s\ (F\ X)$—are allowed provided right-hand sides does not restrict the types of the variables more than left-hand sides. Notice that our more permissive behavior comes directly from the definition of well-typed rule and no specific treatment of existential constructors is needed¶, in the same way that the *size* function from Section 1—page 3—has rules whose argument have a more specific type (*bool*, *nat* and $[\alpha]$) than the type for them that comes from the declared type of the function (α).

Apart from existential constructors, in functional logic languages HO-patterns can introduce a similar opacity than existential types. 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. This opacity problem, originally identified in (González-Moreno et al., 2001), is solved in (López-Fraguas et al., 2010) by means of *opaque variables*. Briefly explained, a data variable is opaque in a pattern if the type of the whole pattern does not univocally fix the type of the variable. That is the case of X in the pattern $snd\ X$: from the type $\beta \to \beta$ of the pattern we cannot know univocally the type of X, which indeed can have any type (*bool*, *int*, $[bool]$ ...). The problems that opaque variables generate for type preservation are solved in (López-Fraguas et al., 2010) by forbidding *critical variables* in program rules (data variables appearing in the righ-hand side which are opaque in a pattern of the left-hand side). However, it is known that this solution rejects functions that do not compromise type preservation although they contain critical variables. The program below shows how the system presented here generalizes that from (López-Fraguas et al., 2010), accepting functions containing critical variables:

$$\mathcal{A} \equiv \mathcal{A}_{basic} \oplus \{\ \mathbf{idSnd} : \forall\alpha,\beta.(\alpha \to \alpha) \to (\beta \to \beta),\ \mathbf{f} : \forall\alpha.(\alpha \to \alpha) \to int\ \}$$
$$\mathcal{P} \equiv \{idSnd\ (snd\ X) \to snd\ X,\ f\ (snd\ X) \to length\ [X],\ f\ (snd\ (X : Xs)) \to length\ Xs\}$$

Variables X and Xs are critical in all the rules, so they are rejected by the type system in (López-Fraguas et al., 2010). However, the type system presented here accepts all the rules because they verify the well-typedness criterion: right-hand sides do not restrict the types of the variables more than left-hand sides.

Another remarkable example using HO patterns is given by the well-known translation of higher-order programs to first-order programs (Warren, 1982) often used as a stage of the compilation of functional logic programs—see e.g. (Antoy and Tolmach, 1999; López-Fraguas et al., 2008). In short, this translation introduces a new function symbol @ (to be read as '*apply*'), and then adds calls to @ in some points in the program and appropriate

¶ In contrast to the explicit treatment of existentially quantified variables using Skolem constants.

rules for evaluating it. This latter aspect is interesting here, since those @-rules are not Damas-Milner typeable. The following program contains the @-rules (written in infix notation) for a concrete example with the constructors *z*, *s*, *[ ]*, *(:)* and the functions *length*, *append* and *snd* with the usual types.

$$\begin{aligned}\mathcal{A} \equiv \quad & \mathcal{A}_{basic} \oplus \{\ \mathbf{length} : \forall\alpha.[\alpha] \to nat, \mathbf{append} : \forall\alpha.[\alpha] \to [\alpha] \to [\alpha], \\ & \mathbf{add} : nat \to nat \to nat, @ : \forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta\ \} \\ \mathcal{P} \equiv \{\ \quad & s\ @\ X \to s\ X, (:)\ @\ X \to (:)\ X, ((:)\ X)\ @\ Y \to (X : Y), \\ & append\ @\ X \to append\ X, (append\ X)\ @\ Y \to append\ X\ Y, \\ & snd\ @\ X \to snd\ X, (snd\ X)\ @\ Y \to snd\ X\ Y, length\ @\ X \to length\ X\ \}\end{aligned}$$

These rules use HO-patterns, which is a cause of rejection in many systems. Even if HO-patterns were allowed, the rules for @ would be rejected by a Damas-Milner-like type system. 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 Section 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 and then use it for concrete types via a conversion function. We develop the idea for the *size* example.

This can be done by using GADTs to represent uniformly the applicative structure of expressions—for instance, the *spines* of (Hinze and Löh, 2007)—then defining *size* over that uniform representations, and finally 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 used 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 $toU : \forall\alpha.\alpha \to univ$

$$\begin{array}{ll} toU\ true \to c\ z\ [\,] & toU\ false \to c\ (s\ z)\ [\,] \\ toU\ z \to c\ (s^2\ z)\ [\,] & toU\ (s\ X) \to c\ (s^3\ z)\ [toU\ X] \\ toU\ [\,] \to c\ (s^4\ z)\ [\,] & toU\ (X{:}Xs) \to c\ (s^5\ z)\ [toU\ X, toU\ Xs] \end{array}$$

(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 Section 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[‖].

[‖] 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.

Admittedly, the type *univ* has less representation power than the spines of (Hinze and Löh, 2007), which could be a better option in more complex situations. Nevertheless, notice that the GADT-based encoding of spines is also valid in 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 (Lux, 2008) for some known problems for the latter—or GADTs, which have a specific type system more complex than Damas-Milner.

Liberality (continued from Section 4). We recall the example of *size*, where our system accepts the expression *size e* as well-typed, 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 types, 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 principal types exist and can be inferred, our definition of well-typed program (Definition 3.1) 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 (Cheney and Hinze, 2003; Schrijvers et al., 2009). 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 (Wadler, 1989) 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 recast as "the parametricity theorem", which basically exploits the fact that the polymorphic type variables in the types of function symbols cannot be instantiated in the left-hand side of program rules. Parametricity was originally developed for the polymorphic λ-calculus, which in particular enjoys the strong normalisation property, so its application to actual languages with practical features like unbounded recursion or partial functions has to be done with care. This can

be easily understood by considering the first example in (Wadler, 1989), stating that for any function $f : \forall\alpha.[\alpha] \to [\alpha]$ and any function g with some (irrelevant) type then $(map\ g) \circ f \equiv f \circ (map\ g)$. The intuition is that, as by parametricity f cannot inspect the polymorphic elements of its input list—to do so it should instantiate the type variable α into a more concrete type in the left-hand side of some program rule for f—then it may only return a rearrangement of that list, maybe dropping or duplicating some of its elements but never introducing new elements. This is not the case for a practical language like Haskell, for example, as we can define the functions $\{loop \to loop, fail \to head\ [\,]\}$, both with type $\forall\alpha.\alpha$, that can be used to introduce new elements in the resulting list for f thus breaking that free theorem (Seidel and Voigtländer, 2010). Similarly an impure feature like Haskell's *seq* operator weakens parametricity because it essentially inspects its polymorphic first argument in order to force its evaluation (Hudak et al., 2007). Nevertheless free theorems can be weakened with several additional conditions so they actually hold for Haskell (Wadler, 1989; Johann and Voigtländer, 2004). These efforts are motivated by the fact that parametricity is used to justify the soundness of some important compiler optimizations, like the "short-cut deforestation" of GHC (GHC-Team, 2011)—although it is admitted that *seq* still makes this particular transformation unsound (Hudak et al., 2007).
Regarding FLP, it is known that non-determinism not only breaks free theorems but also equational rules for concrete functions that hold for Haskell, like $(filter\ p) \circ (map\ h) \equiv (map\ h) \circ (filter\ (p \circ h))$ (Christiansen et al., 2010). The situation gets even worse when considering extra variables and narrowing—not treated in the present work but standard in FLP systems—because then the function f above could also introduce a free variable in its resulting list, thus breaking the equivalence from a new side wrt. Haskell, as in FLP free variables may produce interesting values in contrast to $loop$ and $fail$.

With our type system, not only those free theorems derived from parametricity are broken, but it is the more fundamental notion of parametricity they rely on that is lost, because functions are allowed to inspect any argument subexpression, as seen in the *size* function from page 3. This has a limited impact in the FLP setting, as free theorems were already heavily compromised by non-determinism and free variables, but it could limit the applicability of our type system to pure FP. For example, working without the hypothesis of parametricity would be a problem for GHC because of its representation of datatypes, which results in an unpredictable behaviour when matching two expressions with different types—as can be seen by using the polymorphic casting function from (Hudak et al., 2007). Fortunately, state-of-the-art FLP systems are based on a compilation to Prolog for which those heterogeneous matchings pose no problem. In fact ours would not be the first type system for FP that allows that kind of liberalized inspections, i.e. it is possible to do that by using GADTs, as seen in Figure 4-b. Nevertheless GADTs—at least those implemented by GHC—are only able to inspect "liberalized" arguments whose type has been already sufficiently refined in the left-to-right Haskell matching process. For example if we interchange the first and third argument of *eq* in Figure 4-b then the program would be rejected by GHC—while it is still accepted by our type system. The reason is that GHC's matching process proceeds from left to right and, as GADT arguments fix their polymorphic types when matched thus fixing the types of the arguments

they liberalize, that ensures the absence of dangerous matchings in GHC. Similarly, classical existential types use skolem constants to forbid liberalized inspections that would threaten parametricity and turn GHC style matching and datatypes representation into an unsound procedure. However, that liberalized inspections just result from the kind of matchings exploited by our liberal functions, therefore the possible application of our type system to concrete Haskell implementations remains an open problem. Maybe a modification of our proposed type system, that would restrict liberal typing of functions to some fragments of the program only, would still enjoy some relevant parametricity property. We consider this an interesting subject of future work.

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 a variety of existential types or GADT-like encodings, and to open new possibilities to genericity; good formal properties (type soundness, protection against unsafe use of HO-patterns, maximal liberality while fulfilling the previous conditions). Regarding the practical interest of our work, we stress the fact that no existing FLP system supports any of the examples in Section 5, in particular the examples of the *equality*—where known problems of *opaque decomposition* (González-Moreno et al., 2001) can be addressed—and *apply* functions, which play important roles in the FLP setting. Moreover, our work provides a valuable alternative to our previous results (González-Moreno et al., 2001; López-Fraguas et al., 2010) about safe uses of HO-patterns. However, considering also the weaknesses discussed in Section 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 programmers can choose the feature—ordinary types, classes, GADTs or our more direct generic functions—that best fits their needs of genericity and/or control in each specific situation.

Some steps to achieve this scenario have been already performed. The first one is a web interface (`http://gpd.sip.ucm.es/LiberalTyping`) of the type system which checks program well-typedness. This web interface supports GADT syntax for data declarations, so all the examples in this paper can be checked. Another performed step is the development of a branch of Toy using the type system proposed in this paper, which can be downloaded at `http://gpd.sip.ucm.es/Toy2Liberal`. This branch lacks syntax for GADT data declaration, however it provides the users a complete and functional Toy system where programs can be compiled and evaluated.

Apart from further implementation work, we consider several lines of future 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. A first step towards the

specification of the translation of type classes has been already developed in (Martin-Martin, 2011).
— Despite of the lack of principal types, some work on type inference can be done, in the spirit of (Schrijvers et al., 2009).
— Combining our genericity with the existence of modules could require adopting *open* types and functions (Löh and Hinze, 2006).
— Narrowing, which poses specific problems to types, should be also considered.

Acknowledgments We are grateful to the anonymous reviewers for suggesting us the idea of an alternative declarative formulation of well-typedness, that lead us to include Definition 4.1 and Proposition 4.1. We also thank Philip Wadler and the rest of reviewers of the previous conference version of the paper, for their stimulating criticisms and comments.

References

Antoy, S. and Tolmach, A. P. (1999). Typed higher-order narrowing without higher-order strategies. In *4th International Symposium on Functional and Logic Programming (FLOPS '09)*, pages 335–353. Springer *LNCS* 1722.

Cheney, J. and Hinze, R. (2003). First-class phantom types. Technical Report TR2003-1901, Cornell University.

Christiansen, J., Seidel, D., and Voigtländer, J. (2010). Free theorems for functional logic programs. In *4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV '10)*, pages 39–48. ACM.

Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*, pages 207–212. ACM.

GHC-Team (2011). The Glorious Glasgow Haskell Compilation System User's Guide. `http://www.haskell.org/ghc/docs/latest/html/users_guide`.

González-Moreno, J., Hortalá-González, T., López-Fraguas, F., and Rodríguez-Artalejo, M. (1999). An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87.

González-Moreno, J., Hortalá-González, T., and Rodríguez-Artalejo, M. (1997). A higher order rewriting logic for functional logic programming. In *14th International Conference on Logic Programming (ICLP '97)*, pages 153–167. MIT Press.

González-Moreno, J., Hortalá-González, T., and Rodríguez-Artalejo, M. (2001). Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1).

Hall, C. V., Hammond, K., Peyton Jones, S., and Wadler, P. (1996). Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138.

Hanus, M. (2007). Multi-paradigm declarative languages. In *23rd International Conference on Logic Programming (ICLP '07)*, pages 45–75. Springer *LNCS* 4670.

Hanus (ed.), M. (2006). Curry: An integrated functional logic language. `http://www.informatik.uni-kiel.de/~curry/report.html`.

Hinze, R. (2006). Generics for the masses. *Journal of Functional Programming*, 16(4-5):451–483.

Hinze, R. and Löh, A. (2007). Generic programming, now! In *Datatype-Generic Programming 2006*, pages 150–208. Springer *LNCS* 4719.

Hudak, P., Hughes, J., Peyton Jones, S., and Wadler, P. (2007). A history of Haskell: being lazy with class. In *3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL '07)*, pages 12–1–12–55. ACM.

Johann, P. and Voigtländer, J. (2004). Free theorems in the presence of *seq*. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 99–110. ACM.

Läufer, K. and Odersky, M. (1994). Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16:1411–1430.

Löh, A. and Hinze, R. (2006). Open data types and open functions. In *8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '06)*, pages 133–144. ACM.

López-Fraguas, F., Martin-Martin, E., and Rodríguez-Hortalá, J. (2010). Liberal Typing for Functional Logic Programs. In *8th Asian Symposium on Programming Languages and Systems (APLAS '10)*, pages 80–96. Springer *LNCS* 6461.

López-Fraguas, F., Martin-Martin, E., and Rodríguez-Hortalá, J. (2010). New results on type systems for functional logic programming. In *18th International Workshop on Functional and Constraint Logic Programming (WFLP '09)*, pages 128–144. Springer *LNCS* 5979.

López-Fraguas, F., Rodríguez-Hortalá, J., and Sánchez-Hernández, J. (2008). Rewriting and call-time choice: the HO case. In *9th International Symposium on Functional and Logic Programming (FLOPS '08)*, pages 147–162. Springer *LNCS* 4989.

López-Fraguas, F. and Sánchez-Hernández, J. (1999). $\mathcal{TOY}$: A multiparadigm declarative system. In *10th Rewriting Techniques and Applications (RTA '99)*, pages 244–247. Springer *LNCS* 1631.

Lux, W. (2008). Adding Haskell-style overloading to Curry. In *Workshop of Working Group 2.1.4 of the German Computing Science Association GI*, pages 67–76.

Lux, W. (2009). Type-classes and call-time choice vs. run-time choice - Post to the Curry mailing list. `http://www.informatik.uni-kiel.de/~curry/listarchive/0790.html`.

Martin-Martin, E. (2009). Advances in type systems for functional logic programming. Master's thesis, Universidad Complutense de Madrid. `http://gpd.sip.ucm.es/enrique/publications/master/masterThesis.pdf`.

Martin-Martin, E. (2011). Type classes in functional logic programming. In *20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '11)*, pages 121–130. ACM.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Mitchell, J. C. and Plotkin, G. D. (1988). Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502.

Moreno-Navarro, J., Mariño, J., del Pozo-Pietro, A., Herranz-Nieva, A., and García-Martín, J. (1996). Adding type classes to functional-logic languages. In *Joint Conference on Declarative Programming, APPIA-GULP-PRODE'96*, pages 427–438.

Perry, N. (1991). *The Implementation of Practical Functional Programming Languages*. Ph.D. thesis, Imperial College, London.

Peyton Jones, S., Vytiniotis, D., and Weirich, S. (2006). Simple unification-based type inference for GADTs (Technical Appendix). Technical Report MS-CIS-05-22, University of Pennsylvania.

Sánchez-Hernández, J. (2006). Constructive failure in functional-logic programming: From theory to implementation. *Journal of Universal Computer Science*, 12(11):1574–1593.

Schrijvers, T., Peyton Jones, S., Sulzmann, M., and Vytiniotis, D. (2009). Complete and decidable type inference for GADTs. In *14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*, pages 341–352. ACM.

Seidel, D. and Voigtländer, J. (2010). Automatically generating counterexamples to naive free theorems. In *10th International Symposium on Functional and Logic Programming (FLOPS '10)*, pages 175–190. Springer LNCS 6009.

Thatté, S. R. (1994). Semantics of type classes revisited. In *8th ACM Conference on LISP and Functional Programming (LFP '94)*, pages 208–219. ACM.

van Bakel, S. and Fernández, M. (1997). Normalization results for typeable rewrite systems. *Information and Computation*, 133(2):73 – 116.

Wadler, P. (1989). Theorems for free! In *4th International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, pages 347–359. ACM.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*, pages 60–76. ACM.

Warren, D. H. (1982). Higher-order extensions to prolog: are they needed? In *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd.

Wright, A. K. and Felleisen, M. (1992). A Syntactic Approach to Type Soundness. *Information and Computation*, 115:38–94.

Appendix A. Proofs and auxiliary results

This appendix contains complete proofs for all the results in the paper. 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 $vran(\pi) = \bigcup_{\alpha \in dom(\pi)} ftv(\alpha\pi)$

b) Provided the domains of two type substitutions π_1 and π_2 are disjoint, 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 A.1 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 A.1 (Soundness of $\Vdash$). $\mathcal{A} \Vdash e : \tau | \pi \Longrightarrow \mathcal{A}\pi \vdash e : \tau$

Theorem A.2 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 more general.

Theorem A.2 (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 A.3 (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 $\mathcal{A} \vdash e : \tau \Longleftrightarrow \mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$, for any σ.
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 A.1, A.2 and A.3 appears in Enrique Martin-Martin's master thesis (Martin-Martin, 2009). □

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

Explanation. If it is possible to derive a type for e with the assumptions $\mathcal{A}$, then the inference will not need to instantiate $\mathcal{A}$. Since $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})[\overline{\alpha_n/\tau_n}] \vdash e : \tau$ then by Theorem A.2 we know that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash e : \tau' | \pi$ and $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi\pi'' = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})[\overline{\alpha_n/\tau_n}]$ for some substitution π''. Therefore $\mathcal{A}\pi\pi'' = \mathcal{A}[\overline{\alpha_n/\tau_n}] = \mathcal{A}$, so π only replace variables in $\mathcal{A}$ which are restored by π''. These replacements are generated by unification steps that substitute free type variables in $\mathcal{A}$ for fresh type variables created during inference. Then we can assume that in these cases unification only replaces fresh variables, obtaining that $\mathcal{A}\pi = \mathcal{A}$. □

A.2. *Proof of Theorem 3.1*

Theorem 3.1
If $wt^{old}_{\mathcal{A}}(\mathcal{P})$ then $wt_{\mathcal{A}}(\mathcal{P})$.

Proof. In (López-Fraguas et al., 2010) and also in 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:

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

and $critVar_{\mathcal{A}}(\lambda t.e) = \emptyset$, i.e., $opaqueVar_{\mathcal{A}}(t) \cap fv(e) = \emptyset$. 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 A.2 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 A.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 we had $\mathcal{A}\pi_t\pi_t'' = \mathcal{A}$ and by Remark A.1 $\mathcal{A}\pi_t = \mathcal{A}$, so $\mathcal{A}\pi_t'' = \mathcal{A}$. Since $\tau_t' \to \tau_e'$ is a variant of $\mathcal{A}(f)$ it contains only type variables which are free in $\mathcal{A}$ or fresh type variables, so π_t'' will not affect it. Defining π_u as $\pi_t''|_{ftv(\tau_t)} + [\gamma/\tau_e']$ we have an unifier, since:

$$\begin{array}{lll}
 & \underline{(\tau_t' \to \tau_e')\pi_t\pi_u} & \pi_t \text{ does not affect } \tau_t' \to \tau_e' \\
= & \underline{(\tau_t' \to \tau_e')\pi_u} & \gamma \notin ftv(\tau_t' \to \tau_e') \\
= & (\tau_t' \to \tau_e')\underline{\pi_t''|_{ftv(\tau_t)}} & \pi_t''|_{ftv(\tau_t)} \text{ does not affect } \tau_t' \to \tau_e' \\
= & \tau_t' \to \underline{\tau_e'} & \text{definition of } \pi_u \\
= & \underline{\tau_t'} \to \gamma\pi_u & \text{Theorem A.2: } \tau_t\pi_t'' = \tau_t' \\
= & \underline{\tau_t\pi_t''|_{ftv(\tau_t)}} \to \gamma\pi_u & \gamma \notin ftv(\tau_t) \\
= & \underline{\tau_t\pi_u \to \gamma\pi_u} & \text{application of substitution} \\
= & (\tau_t \to \gamma)\pi_u &
\end{array}$$

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 A.2 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 A.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:

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

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 A.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.3. *Proof of Theorem 4.1: Progress*

Theorem 4.1 (Progress)
If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$ and e is ground, then either e is a pattern or $\exists e'.\ \mathcal{P} \vdash e \rightarrowtail e'$.

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 $e \equiv fail$.
$f \in FS^n$)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 $(f \rightarrow e) \in \mathcal{P}$ then we can apply rule (Fapp), so $\mathcal{P} \vdash s \rightarrowtail e$. Otherwise there is not any rule $(l \rightarrow e') \in \mathcal{P}$ such that l and f unify, so we can apply the rule for the matching failure (Ffail) obtaining $\mathcal{P} \vdash f \rightarrowtail fail$.

Inductive Step

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

$$\textbf{[APP]}\frac{\begin{array}{c}\mathcal{A} \vdash e_1 : \tau_1 \to \tau \\ \mathcal{A} \vdash e_2 : \tau_1\end{array}}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

Both e_1 and e_2 are well-typed and ground. If e_1 is not a pattern, by the Induction Hypothesis we have $\mathcal{P} \vdash e_1 \rightarrowtail e_1'$ and using the (Contx) rule we obtain $\mathcal{P} \vdash e_1\ e_2 \rightarrowtail e_1'\ e_2$. 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 $\mathcal{P} \vdash \mathit{fail}\ e_2 \rightarrowtail \mathit{fail}$.
    • 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 $\mathcal{A} \vdash e_1\ e_2 : \tau$, and Lemma A.2 states that *junk* expressions cannot be well-typed wrt. any 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 $\mathcal{P} \vdash e_1\ e_2 \rightarrowtail r\theta$. 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. Therefore we can apply the rule for the matching failure (Ffail) obtaining $\mathcal{P} \vdash e_1\ e_2 \rightarrowtail \mathit{fail}$.

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

$$\textbf{[LET]}\frac{\begin{array}{c}\mathcal{A} \vdash e_1 : \tau_X \\ \mathcal{A} \oplus \{X : Gen(\tau_X, \mathcal{A})\} \vdash e_2 : \tau\end{array}}{\mathcal{A} \vdash \mathit{let}\ X = e_1\ \mathit{in}\ e_2 : \tau}$$

There are two cases depending on whether e_1 is a pattern or not:

— e_1 is a pattern) Then we can use the (Bind) rule, obtaining $\mathcal{P} \vdash \mathit{let}\ X = e_1\ \mathit{in}\ e_2 \rightarrowtail e_2[X/e_1]$.
— e_1 is not a pattern) Since $\mathit{let}\ X = e_1\ \mathit{in}\ e_2$ 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': $\mathcal{P} \vdash e_1 \rightarrowtail e_1'$. Using the (Contx) rule we can transform this local step into a step in the whole expression: $\mathcal{P} \vdash \mathit{let}\ X = e_1\ \mathit{in}\ e_2 \rightarrowtail \mathit{let}\ X = e_1'\ \mathit{in}\ e_2$.

□

A.4. *Proof of Theorem 4.2: Type Preservation*

Theorem 4.2 (Type Preservation)
If $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ *and* $\mathcal{P} \vdash e \rightarrowtail e'$, *then* $\mathcal{A} \vdash e' : \tau$.

Proof. We proceed by case distinction over the rule of the *let*-rewriting relation $\rightarrowtail$ (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 $f\ t_1\theta \ldots t_m\theta$ (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 $wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to r)$, and by the definition of well-typed rule (Definition 3.2) 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)* $\mathcal{A} \vdash t'_i : \tau_i$. Notice that these $\overline{\tau_n}$ are unique as the left-hand side of the rule is linear, so each t'_i will appear once.

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 A.2 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 A.1):

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

Using the fact that type derivations are closed under substitutions (Theorem A.3-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 A.3-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 A.3-b) we can add the type assumptions $\{\overline{X_n : \tau_n}\}$ to the type derivations in *(F)*, obtaining *(Q)* $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t'_i : \tau_i$. Notice that we assume that $\overline{X_n}$ do not appear in $t'_i \equiv X_i\theta$, as $\overline{X_n}$ are the variables of the rule.

By Theorem A.3-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)*:

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

Finally, the data variables $\overline{X_n}$ do not appear in $r\theta$, so by Theorem A.3-b) we can erase that assumptions in *(R)*:

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

(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 (Martin-Martin, 2009).

□

A.5. *Proof of Theorem 4.3: Maximal liberality of well-typedness conditions*

In order to prove Theorem 4.3 we will use an auxiliary result relating the types involved in type derivations to the types inferred by a type inference:

Lemma A.1. Given a closed set of assumptions $\mathcal{A}$, if $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash e : \tau_g|\pi_g$ and $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ (for some $\overline{\alpha_n}$ fresh) then there exists some π such that $\tau_g\pi = \tau$ and $\alpha_i\pi_g\pi = \tau_i$ for every $i \in [1..n]$.

Proof. Straightforward by Theorem A.2 with $\pi' \equiv [\overline{\alpha_n/\tau_n}]$. □

Theorem 4.3 (Maximal liberality of well-typedness conditions)
Let $\mathcal{A}$ be a closed set of assumptions, and assume that $\mathcal{P}$ is a program which is not well-typed wrt. $\mathcal{A}$, but such that every rule $R \in \mathcal{P}$ verifies the condition i) of well-typedness in Definition 3.2. Then there exists a rule $(f\ t_1 \ldots t_m \to e) \in \mathcal{P}$ with variables $\overline{X_n}$ and there exist types $\overline{\tau_n}$, τ such that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1 \ldots t_m : \tau$ and $f\ t_1 \ldots t_m \rightarrowtail e$ but $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \nvdash e : \tau$.

Proof. For every rule, *i)* holds by hypothesis and *iv)* holds trivially as $\mathcal{A}$ is closed. Therefore either condition *ii)* or *iii)* must fail for some rule $R \equiv (f\ t_1 \ldots t_m \to e) \in \mathcal{P}$. The condition *i)* says that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t_1 \ldots t_m : \tau_L|\pi_L$, for some τ_L, π_L. Then, by the soundness of $\Vdash$ (Theorem A.1) we have

$$(1) \quad \mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_L}\} \vdash f\ t_1 \ldots t_m : \tau_L$$

Moreover, using (Fapp) and the rule R it is possible to perform the rewrite step

$$(2) \quad f\ t_1 \ldots t_m \rightarrowtail_{(Fapp)} e$$

We will now see that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n \pi_L}\} \not\vdash e : \tau_L$, which will finish the proof by taking $\overline{\tau_n} = \overline{\alpha_n \pi_L}$ and $\tau = \tau_L$. We distinguish two cases depending on which of the conditions *ii*) or *iii*) in Definition 3.2 fails for the rule R.

a) If *ii*) does not hold for R then by the completeness of $\Vdash$ (Theorem A.2) there are not any types $\overline{\tau_n}$, τ such that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$, so in particular $\mathcal{A} \oplus \{\overline{X_n : \alpha_n \pi_L}\} \not\vdash e : \tau_L$ as desired.

b) If *ii*) holds but *iii*) does not, then we have that there exist some τ_R, π_R such that

$$(3) \quad \mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_R | \pi_R \quad \text{by } ii)$$

$$(4) \quad \neg \exists \pi . (\tau_L, \overline{\alpha_n \pi_L}) = (\tau_R, \overline{\beta_n \pi_R}) \pi \quad \text{by failure of } iii)$$

Condition (4) is equivalent to say that

$$(5) \quad \forall \pi . (\tau_L = \tau_R \pi \Longrightarrow \exists i \in [1..n] . \alpha_i \pi_L \neq \beta_i \pi_R \pi)$$

We reason now by contradiction, assuming that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n \pi_L}\} \vdash e : \tau_L$ (we want to prove the contrary). Then by (3) and Lemma A.1 we have that there is some π such that $\tau_R \pi = \tau_L$ and $\beta_i \pi_R \pi = \alpha_i \pi_L$ for every $i \in [1..n]$, which contradicts (5).

□

The previous proof is constructive since it shows that given a rule $(f\ t_1 \ldots t_m \to e) \in \mathcal{P}$ not holding *ii*) or *iii*), the evaluation step $f\ t_1 \ldots t_m \rightarrowtail_{(Fapp)} e$ never preserves types using $\overline{\tau_n} = \overline{\alpha_n \pi_L}$ and $\tau = \tau_L$.

The following examples illustrates the lost of type preservation in the different cases considered in the proof. The rule $f_1 \to not\ [\,]$ with assumption $f_1 : bool$ does not verify point *ii*) since the right-hand side is ill-typed. In this case it is easy to check that $\mathcal{A} \vdash f_1 : bool$ and $f_1 \rightarrowtail not\ [\,]$, but $\mathcal{A} \not\vdash not\ [\,] : bool$—indeed, $not\ [\,]$ does not have any type. The rule $f_2 \to true$ with assumption $f_2 : nat$ verifies point *ii*) but not *iii*) because *bool* does not match *nat*, which corresponds to the case when (5) holds because the antecedent in the implication always fails. Trivially $\mathcal{A} \vdash f_2 : nat$ and $f_2 \rightarrowtail true$, but $\mathcal{A} \not\vdash true : nat$. Finally, the rule $f_3\ X \to not\ X$ with assumption $f_3 : \forall \alpha . \alpha \to bool$ illustrates the case when point *ii*) holds but *iii*) does not, although in this case the antecedent $\tau_L = \tau_R \pi$ of (5) holds for some π (for any π indeed, since $\tau_L = \tau_R = bool$). What happens here is that the type *bool* inferred for the variable X in the right-hand side does not match the type α inferred in the left-hand side. In this case it is clear that $\mathcal{A} \oplus \{X : \alpha\} \vdash f_3\ X : bool$ and $f_3\ X \rightarrowtail not\ X$, but $\mathcal{A} \oplus \{X : \alpha\} \not\vdash not\ X : bool$.

A.6. *Proof of Proposition 4.1*

Proposition 4.1
Consider a type-complete set of assumptions $\mathcal{A}$, and a program rule $R \equiv f\ t_1 \ldots t_m \to e$. Then R preserves types iff $wt_{\mathcal{A}}(R)$.

Proof.

$\Longrightarrow$) We proceed proving the contrapositive

$$\neg wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e) \Longrightarrow f\ t_1 \ldots t_m \to e \text{ is not type-preserving}$$

If $f\ t_1 \ldots t_m \to e$ is not well-typed because it does not verify point i) of Definition 3.2 then by completeness of type inference (Theorem A.2) the left-hand side of the rule does not admit any type, so the rule is not type-preserving.
If $f\ t_1 \ldots t_m \to e$ is not well-typed but it verifies the point point i) of Definition 3.2 then by soundness of type inference (Theorem A.1) its left-hand side admits some—point i) of Definition 4.1 of type-preserving rule. In this case we have to prove that

$$\neg wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e) \Longrightarrow \exists\theta.(\mathcal{A} \vdash f\ t_1\theta \ldots t_m\theta : \tau \wedge \mathcal{A} \not\vdash e\theta : \tau)$$

As the the point i) of the definition of well-typed rule is verified, by Theorem 4.3 (maximal liberality of well-typedness conditions) we know that there are types $\overline{\tau_n}$ and τ such that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1 \ldots t_m : \tau$ and $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \not\vdash e : \tau$. The set of assumptions $\mathcal{A}$ is type-complete, so there are patterns $\overline{t_{\tau_n}}$ which can only have those types, i.e., $\mathcal{A} \vdash t_{\tau_i} : \tau_i$. As the variables $\overline{X_n}$ are the variables of the rule we can assume that they do not appear in the patterns $\overline{t_{\tau_n}}$, so by Theorem A.3-b) we have that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t_{\tau_i} : \tau_i$. Using Theorem A.3-c) we can replace the variables $\overline{X_n}$ in $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1 \ldots t_m : \tau$ by the patterns of the same type with the substitution $\theta \equiv [\overline{X_n/t_{\tau_n}}]$, obtaining $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1\theta \ldots t_m\theta : \tau$. Again by Theorem A.3-b) we can remove the variables $\overline{X_n}$ from the set of assumptions as they do not ocur in $f\ t_1\theta \ldots t_m\theta$, obtaining $\mathcal{A} \vdash f\ t_1\theta \ldots t_m\theta : \tau$. On the other hand, it is easy to check that $\mathcal{A} \not\vdash e\theta : \tau$ because $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \not\vdash e : \tau$ and we replace the variables $\overline{X_n}$ by patterns $\overline{t_{\tau_n}}$ which can only have those types $\overline{\tau_n}$.
$\Longleftarrow$) If $wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e)$ then from point i) of Definition 3.2 (well-typed rule) and Theorem A.1 (soundness of type inference), the left-hand side of the rule admits some type—point i) of Definition 4.1 (type-preserving rule). Regarding the point ii) of Definition 4.1, consider an arbitrary θ and τ such that $\mathcal{A} \vdash f\ t_1\theta \ldots t_m\theta : \tau$. Following the same reasoning as in the proof for the (Fapp) rule in Theorem 4.2 (type preservation) we conclude that $\mathcal{A} \vdash e\theta : \tau$.

□

A.7. *Proof of Lemma 4.1: Faulty Expressions are ill-typed*

In order to prove Lemma 4.1 we use an auxiliary result stating that *junk* expressions cannot have a valid type wrt. any set of assumptions $\mathcal{A}$:

Lemma A.2. If e is a *junk* expression then there is no $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$.

Proof. By contradiction. Assume there is $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$. If e is *junk* then it has the form $c\ t_1 \ldots t_n$ with $c \in CS^m$ and $n > m$, i.e., $(c\ t_1 \ldots t_m)\ t_{m+1} \ldots t_n$. The type

derivation for e must contain a subderivation of the form:

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

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 is not a functional type $(\tau_1 \to \tau)$, so we have found a contradiction. □

Using the previous result, we can prove Lemma 4.1:

Lemma 4.1 (Faulty Expressions are ill-typed)
If e is faulty then there is no $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$.

Proof. We prove it by contradiction. Suppose that e has a junk subexpression e' and $\mathcal{A} \vdash e : \tau$. Therefore, in that derivation we have a subderivation $\mathcal{A}' \vdash e' : \tau'$ (for some $\mathcal{A}'$ and τ'). By Lemma A.2 those $\mathcal{A}'$ and τ' cannot exist, so we have found a contradiction. □

A.8. *Proof of Theorem 4.4: Syntactic Soundness*

We need some auxiliary results:

Lemma A.3 (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 (Theorem 4.1).

□

Lemma A.4. If $\mathcal{P} \vdash e \rightarrowtail e'$ and $\mathcal{P}$ does not contains extra variables in its rules, then $fv(e') \subseteq fv(e)$.

Proof. Easily by case distinction over the rule applied in the step $\mathcal{P} \vdash e \rightarrowtail e'$.

□

From the previous lemma follows an useful corollary:

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

Using the previous results, the proof of Theorem 4.4 is straightforward:

Theorem 4.4 (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 $\mathcal{A} \vdash e' : \tau$.*

Proof. Let e' be an arbitrary expression in $nf_{\mathcal{P}}(e)$. Since e is ground, by Corollary A.1 e' is also ground. Applying Type Preservation (Theorem 4.2) in all the reduction steps we have $\mathcal{A} \vdash e' : \tau$. Since e' is a well-typed normal form, by Lemma A.3 e' is a pattern. □