Liberal Typing for Functional Logic Programs*

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

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

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

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

1 Introduction

Functional logic programming. Functional logic languages [9] like TOY [19] or Curry [10] have a strong resemblance to lazy functional languages like Haskell [13]. A remarkable difference is that functional logic programs (FLP) can be non-confluent, giving raise to so-called *non-deterministic functions*, for which a *call-time choice* semantics [6] is adopted. The following program is a simple example, using natural numbers given by the constructors z and s—we follow syntactic conventions of some functional logic languages where function and constructor names are lowercased, and variables are uppercased—and assuming a natural definition for *add*: $\{\ f\ X \to X,\ f\ X \to s\ X,\ double\ X \to add\ X\ X\ \}$. Here, f is non-deterministic ($f\ z$ evaluates both to z and $s\ z$) and, according to call-time choice, *double (f z)* evaluates to z and *s (s z)* but not to $s\ z$. Operationally, call-time choice means that all copies of a non-deterministic subexpression ($f\ z$ in the example) created during reduction share the same value.

In the HO-CRWL[1] approach to FLP [7], followed by the TOY system, programs can use *HO-patterns* (essentially, partial applications of symbols to other

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

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

K. Ueda (Ed.): APLAS 2010, LNCS 6461, pp. 80–96, 2010.

patterns) in left hand sides of function definitions. This corresponds to an *intensional* view of functions, i.e., different descriptions of the same 'extensional' function can be distinguished by the semantics. This is not an exoticism: it is known [18] that extensionality is not a valid principle within the combination of HO, non-determinism and call-time choice. It is also known that *HO-patterns* cause some bad interferences with types: [8] and [17] considered that problem, and this paper improves on those results.

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

Types, FLP and genericity. FLP languages are typed languages adopting classical Damas-Milner types [5]. However, their treatment of types is very simple, far away from the impressive set of possibilities offered by functional languages like Haskell: type and constructor classes, existential types, GADTs, generic programming, arbitrary-rank polymorphism ... Some exceptions to this fact are some preliminary proposals for type classes in FLP [23,20], 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 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 [26] were invented to deal with those situations. Some further developments of the idea of generic programming [11] are based on type classes, while others [12] have preferred to use simpler extensions of Damas-Milner system, such as GADTs [3,25]. We propose a modification of Damas-Milner type system that accepts natural definitions of intrinsically generic functions like equality. The following example illustrates the main points of our approach.

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

size true → *s z* *size false* → *s z*
size z → *s z* *size (s X)* → *s (size X)*
size nil → *s z* *size (cons X Xs)* → *s (add (size X) (size Xs))*

However, as far as *bool*, *nat* and $[\alpha]$ are different types, this program would be rejected as ill-typed in a language using Damas-Milner system, since we obtain contradictory types for different rules of *size*. This is a typical case where one wants some support for genericity. Type classes certainly solve the problem if you define a class *Sizeable* and declare *bool, nat* and $[\alpha]$ as instances of it. GADT-based solutions would add an explicit representation of types to the encoding of *size* converting it into a so-called *type-indexed* function [12]. This kind of encoding is also supported by our system (see the *show* function in

Ex. 1 and *eq* in Fig 4-b later), but the interesting point is that our approach allows also a simpler solution: the program above becomes well-typed in our system simply by declaring *size* to have the type $\forall\alpha.\alpha \to nat$, of which each rule of *size* gives a more concrete instance. A detailed discussion of the advantages and disadvantages of such liberal declarations appears in Sect. 6 (see also Sect. 4).

The proposed well-typedness criterion requires only a quite simple additional check over usual type inference for expressions, but here 'simple' does not mean 'naive'. Imposing the type of each function rule to be an instance of the declared type is a too weak requirement, leading easily to type unsafety. As an example, consider the rule $f\ X \to not\ X$ with the assumptions $f : \forall\alpha.\alpha \to bool$, $not : bool \to bool$. The type of the rule is $bool \to bool$, which is an instance of the type declared for f. However, that rule does not preserve the type: the expression $f\ z$ is well-typed according to f's declared type, but reduces to the ill-typed expression $not\ z$. Our notion of well-typedness, roughly explained, requires also that right-hand sides of rules do not restrict the types of variables more than left-hand sides, a condition that is violated in the rule for f above. Def. 1 in Sect. 3.3 states that point with precision, and allows us to prove type soundness for our system.

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

- After some preliminaries, in Sect. 3 we present a novel notion of well-typed program for FLP that induces a simple and direct way of programming type-indexed and generic functions. The approach supports also existential types, opaque HO-patterns and GADT-like encodings, not available in current FLP systems.
- Sect. 4 is devoted to the properties of our type system. We prove that well-typed programs enjoy *type preservation*, an essential property for a type system; then by introducing *failure* rules to the formal operational calculus, we also are able to ensure the *progress* property of well-typed expressions. Based on those results we state type soundness. Complete proofs can be found in [16].
- In Sect. 5 we give a significant collection of examples showing the interest of the proposal. These examples cover type-indexed functions, existential types, opaque higher-order patterns and generic functions. None of them is supported by existing FLP systems.
- Our well-typedness criterion goes far beyond the solutions given in previous works [8,17] to type-unsoundness problems of the use of *HO-patterns* in function definitions. We can type equality, solving known problems of *opaque decomposition* [8] (Sect. 5.1) and, most remarkably, we can type the *apply* function appearing in the HO-to-FO translation used in standard FLP implementations (Sect. 5.2).
- Finally we discuss in Sect. 6 the strengths and weaknesses of our proposal, and we end up with some conclusions in Sect. 7.

2 Preliminaries

We assume a signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint sets of *data constructor* and *function* symbols resp., all of them with associated arity. We write CS^n (resp. FS^n) for the set of constructor (function) symbols of arity n, and if a symbol h is in CS^n or FS^n we write $ar(h) = n$. We consider a special constructor $fail \in CS^0$ to represent pattern matching failure in programs as it is proposed for GADTs [3,24]. We also assume a denumerable set $\mathcal{DV}$ of *data variables* X. Fig. 1 shows the syntax of *patterns* $\in Pat$—our notion of values—and *expressions* $\in Exp$. We split the set of patterns in two: *first order patterns* $FOPat \ni fot ::= X \mid c\ fot_1 \ldots fot_n$ 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 of *free variables* of an expression—$fv(e)$—is defined in the usual way. Notice that since our let expressions do not support recursive definitions the binding of the variable only affect e_2: $fv(let\ X = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus \{X\})$. We say that an expression e is *ground* if $fv(e) = \emptyset$. A *one-hole context* is defined as $\mathcal{C} ::= [] \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C}$. A *data substitution* θ is a finite mapping from data variables to patterns: $[\overline{X_n/t_n}]$. Substitution application over data variables and expressions is defined in the usual way. The empty substitution is written as id. A *program rule* r is defined as $f\ \overline{t_n} \to e$ where the set of patterns $\overline{t_n}$ is linear (there is not repetition of variables), $ar(f) = n$ and $fv(e) \subseteq \bigcup_{i=1}^{n} var(t_i)$. Therefore, extra variables are not considered in this paper. The constructor *fail* is not supposed to occur in the rules, although it does not produce any technical problem. A program $\mathcal{P}$ is a set of program rules: $\{r_1, \ldots, r_n\} (n \geq 0)$.

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 $ar(C) = n$. Fig. 1 shows the syntax of *simple types* and *type-schemes*. The set of *free type variables (ftv)* of a simple type τ is $var(\tau)$, and for type-schemes $ftv(\forall \overline{\alpha_n}.\tau) = ftv(\tau) \setminus \{\overline{\alpha_n}\}$. We say a type-scheme σ is *closed* if $ftv(\sigma) = \emptyset$. A *set of assumptions* $\mathcal{A}$ is $\{\overline{s_n : \sigma_n}\}$, where $s_i \in CS \cup FS \cup \mathcal{DV}$. We require set of assumptions to be *coherent* wrt. CS, i.e., $\mathcal{A}(fail) = \forall \alpha.\alpha$ and for every c in $CS^n \setminus \{fail\}$, $\mathcal{A}(c) = \forall \overline{\alpha}.\tau_1 \to \ldots \to \tau_n \to (C\ \tau'_1 \ldots \tau'_m)$ for some type constructor C with $ar(C) = m$. Therefore the assumptions for constructors must correspond to their arity and, as in [3,24], the constructor *fail* can have any type. The union of sets of assumptions is denoted by $\oplus$: $\mathcal{A} \oplus \mathcal{A}'$ contains all the assumptions in $\mathcal{A}'$ and the assumptions in $\mathcal{A}$ over symbols not appearing in $\mathcal{A}'$. For sets of assumptions $ftv(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. Notice that type-schemes for data constructors may be existential, i.e., they can be of the form $\forall \overline{\alpha_n}.\overline{\tau} \to \tau'$ where $(\bigcup_{\tau_i \in \overline{\tau}} ftv(\tau_i)) \setminus ftv(\tau') \neq \emptyset$. If $(s : \sigma) \in \mathcal{A}$ we write $\mathcal{A}(s) = \sigma$. A *type substitution* π is a finite mapping from type variables to simple types $[\overline{\alpha_n/\tau_n}]$. Application of type substitutions to simple types is defined in the natural way and for type-schemes consists in applying the substitution only to their free variables. This notion is extended to set of assumptions in the obvious way. We

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

Fig. 1. Syntax of expressions and programs

say σ is an *instance* of σ' if $\sigma = \sigma'\pi$ for some π. A simple type τ' is a *generic instance* of $\sigma = \forall\overline{\alpha_n}.\tau$, written $\sigma \succ \tau'$, if $\tau' = \tau[\overline{\alpha_n/\tau_n}]$ for some $\overline{\tau_n}$. Finally, τ' is a *variant* of $\sigma = \forall\overline{\alpha_n}.\tau$, written $\sigma \succ_{var} \tau'$, if $\tau' = \tau[\alpha_n/\beta_n]$ and $\overline{\beta_n}$ are fresh type variables.

3 Formal Setup

3.1 Semantics

The operational semantics of our programs is based on *let*-rewriting [18], a high level notion of reduction step devised to express call-time choice. For this paper, we have extended *let*-rewriting with two rules for managing failure of pattern matching (Fig. 2), playing a role similar to the rules for pattern matching failures in GADTs [3,24]. We write $\to^{lf}$ for the extended relation and $\mathcal{P} \vdash e \to^{lf} e'$ ($\mathcal{P} \vdash e \twoheadrightarrow^{lf} e'$ resp.) to express one step (zero or more steps resp.) of $\to^{lf}$ using

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

(Ffail) $f\ t_1 \ldots t_n \to^{lf} \mathit{fail}$, 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 $f\ t_1 \ldots t_n$ unify

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

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

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

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

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

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

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

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

the program $\mathcal{P}$. By $nf_{\mathcal{P}}(e)$ we denote the set of *normal forms* reachable from e, i.e., $nf_{\mathcal{P}}(e) = \{e' \mid \mathcal{P} \vdash e \twoheadrightarrow^{lf} e' \text{and } e' \text{ is not } \rightarrow^{lf}\text{-reducible}\}$.

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

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

Notice that with the new rules (Ffail) and (FailP) there are still some expressions whose evaluation can get stuck, as happens with *junk expressions* like $true\ z$. As we will see in Sect. 4, this can only happen to ill-typed expressions. We will further discuss there the issues of *fail*-ended and stuck reductions.

3.2 Type Derivation and Inference for Expressions

Both derivation and inference rules are based on those presented in [17]. Our type derivation rules for expressions (Fig. 3-a) correspond to the well-known variation of Damas-Milner's [5] type system with syntax-directed rules, so there is nothing essentially new here—the novelty will come from the notion of well-typed program. $Gen(\tau, \mathcal{A})$ is the closure or generalization of τ wrt. $\mathcal{A}$, which generalizes all the type variables of τ that do not appear free in $\mathcal{A}$. Formally: $Gen(\tau, \mathcal{A}) = \forall \overline{\alpha_n}.\tau \ where\ \{\overline{\alpha_n}\} = ftv(\tau) \setminus ftv(\mathcal{A})$. We say that e is well-typed under $\mathcal{A}$, written $wt_{\mathcal{A}}(e)$, if there exists some τ such that $\mathcal{A} \vdash e : \tau$; otherwise it is ill-typed.

[ID] $\dfrac{}{\mathcal{A} \vdash s : \tau}$ if $\mathcal{A}(s) \succ \tau$

[APP] $\dfrac{\mathcal{A} \vdash e_1 : \tau_1 \rightarrow \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$

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

a) Type derivation rules

[iID] $\dfrac{}{\mathcal{A} \Vdash s : \tau | id}$ if $\mathcal{A}(s) \succ_{var} \tau$

[iAPP] $\dfrac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash e_1\ e_2 : \alpha\pi | \pi_1\pi_2\pi}$ if $\alpha\ fresh \wedge \pi = mgu(\tau_1\pi_2, \tau_2 \rightarrow \alpha)$

[iLET] $\dfrac{\mathcal{A} \Vdash e_1 : \tau_X | \pi_X \quad \mathcal{A}\pi_X \oplus \{X : Gen(\tau_X, \mathcal{A}\pi_X)\} \Vdash e_2 : \tau | \pi}{\mathcal{A} \Vdash \textbf{let}\ X = e_1\ \textbf{in}\ e_2 : \tau | \pi_X\pi}$

b) Type inference rules

Fig. 3. Type system

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

3.3 Well-Typed Programs

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

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

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

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

The first two points check that both right and left hand sides of the rule can have a valid type assigning *some* types for the variables. Furthermore, it obtains the most general types for those variables in both sides. The third point is the most important. It checks that the obtained most general types for the right-hand side and the variables appearing in it are more general than the ones for the left-hand side. This fact guarantees the *type preservation* property (i.e., the expression resulting after a reduction step has the same type as the original one) when applying a program rule. Moreover, this point ensures a correct management of both *skolem* constructors [14] and *opaque variables* [17], either introduced by the presence of existentially quantified constructors or higher order patterns. Finally, the last point guarantees that the set of assumptions is not modified by neither the type inference nor the matching substitution. In practice, this point holds trivially if type assumptions for program functions are closed, as it is usual.

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

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

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

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

The rules for the functions *id* and *snd* are well-typed. The function *unpack* is taken from [8] as a typical example of the type problems that HO-patterns can produce. According to Def. 1 the rule of *unpack* is not well-typed since the tuple $(\tau_L, \overline{\alpha_n \pi_L})$ inferred for the left-hand side is (γ, δ), which is not matched by the tuple (η, η) inferred as $(\tau_R, \overline{\beta_n \pi_R})$ for the right-hand side. This shows the problem of existential type variables that "escape" from the scope. If that rule was well-typed then type preservation could not be granted anymore—e.g. consider the step $unpack\ (snd\ true) \to_{lf} true$, where the type *nat* can be assigned to *unpack* (*snd true*) but *true* can only have type *bool*. The rule for *eq* is well-typed because the tuple inferred for the right-hand side, $(bool, \gamma)$, matches the one inferred for the left-hand side, $(bool, nat)$. In the rule for *show* the inference obtains $([char], nat)$ for both sides of the rule, so it is well-typed.

The functions f and $flist$ show that our type system cannot be forced to accept an arbitrary function definition by generalizing its type assumption. For instance, the first rule for f is not well-typed since the type *nat* inferred for the right-hand side does not match γ, the type inferred for the left-hand side. The second rule for f is also ill-typed for a similar reason. If these rules were well-typed, type preservation would not hold: consider the step $f\ true \to_{lf} z$; $f\ true$ can have any type, in particular *bool*, but z can only have type *nat*. Concerning $flist$, its type assumption cannot be made more general for its first argument: it can be seen that there is no τ 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 some expressions presenting those situations. Examples of this are $flist\ z$ and $eq\ z\ true$, which are ill-typed since the type of the function prevents the existence of program rules that can be used to rewrite these expressions: $flist$ can only have rules treating lists as argument and *eq* can only have rules handling both arguments of the same type.

Def. 1 is based on the notion of type inference of expressions to stress the fact that it can be implemented easily. For each program rule, conditions *i*) and *ii*) use the algorithm of type inference for expressions, *iii*) is just matching, and

iv) holds trivially in practice, as we have noticed before. A more declarative alternative to Def. 1 based on type derivations can be found in [16].

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

In [17] we extended Damas-Milner types with some extra control over HO-patterns, leading to another definition of well-typed programs (we write $wt^{old}_{\mathcal{A}}(\mathcal{P})$ for that). All valid programs in [17] are still valid:

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

To further appreciate the usefulness of the new notion with respect the old one, notice that all the examples in Sect. 5 are rejected as ill-typed by [17].

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 [3,24]. After that, we follow a syntactic approach similar to [28].

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

The *type preservation* result states that in well-typed programs reduction does not change types.

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

In order to follow a syntactic approach similar to [28] we need to define some properties about expressions:

Definition 2. *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, $snd\ (z\ z)\ true \rightarrow^{lf} true$. However all faulty expressions are ill-typed:

Lemma 1 (Faulty Expressions are ill-typed). *If* e *is faulty then there is no* $\mathcal{A}$ *such that* $wt_{\mathcal{A}}(e)$.

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

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

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

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

We discuss now the strength of our results.

- **Progress and type preservation:** In [22] 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 patterns, like Haskell or ours, not all run-time failures should be seen as wrong, as happens with definitions like *head (cons x xs)* $\rightarrow x$, where there is no rule for *(head nil)*. 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 nil* $\rightarrow$ *error* where *error* is a value accepting any type. With it, *(head nil)* reduces to *error* and is not wrong, but *(head true)*, which is ill-typed, is wrong and its reduction gets stuck. In our setting, completing definitions would be more complex because of HO-patterns that could lead to an infinite number of 'missing' cases. Our *failure* rules in Sect. 2 try to play a similar role. We prefer the word *fail* instead of *error* because, in contrast to FP systems where an attempt to evaluate *(head nil)* 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 nil)* 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 $e \rightarrow^* fail$ 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 languages. Consider the function *size* of Sect. 1. For any well-typed 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 nil)*. This can be appreciated as a weakness, and is further discussed in Sect. 6 in connection to type classes and GADT's.
- **Syntactic soundness and faulty expressions:** Th. 4 and 5 are easy consequences of progress and type preservation. Th. 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. Th. 5 says nothing about

them; it is type preservation who ensures that those expressions will not occur in any reduction starting in a well-typed expression. Still, Th. 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 $\mathcal{P}$. In the examples we consider the following initial set of assumptions:

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

5.1 Type-Indexed Functions

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

$\mathcal{A} \equiv \mathcal{A}_{basic} \oplus \{\mathbf{eq} : \forall\alpha.\alpha \to \alpha \to bool\}$
$\mathcal{P} \equiv \{\ eq\ true\ true \to true,$
$eq\ true\ false \to false,$
$eq\ false\ true \to false,$
$eq\ false\ false \to true,$
$eq\ z\ z \to true,$
$eq\ z\ (s\ X) \to false,$
$eq\ (s\ X)\ z \to false,$
$eq\ (s\ X)\ (s\ Y) \to eq\ X\ Y,$
$eq\ (pair\ X_1\ Y_1)\ (pair\ X_2\ Y_2) \to (eq\ X_1\ X_2) \wedge (eq\ Y_1\ Y_2)\ \}$

a) Original program

$\mathcal{A} \equiv \mathcal{A}_{basic} \oplus$
$\{\ \mathbf{eq} : \forall\alpha.repr\ \alpha \to \alpha \to \alpha \to bool,$
$\mathbf{rbool} : repr\ bool, \mathbf{rnat} : repr\ nat,$
$\mathbf{rpair} : \forall\alpha,\beta.repr\ \alpha \to repr\ \beta \to repr\ (pair\ \alpha\ \beta)\ \}$

$\mathcal{P} \equiv \{\ eq\ rbool\ true\ true \to true,$
$eq\ rbool\ true\ false \to false,$
$eq\ rbool\ false\ true \to false,$
$eq\ rbool\ false\ false \to true,$
$eq\ rnat\ z\ z \to true,$
$eq\ rnat\ z\ (s\ X) \to false,$
$eq\ rnat\ (s\ X)\ z \to false,$
$eq\ rnat\ (s\ X)\ (s\ Y) \to eq\ rnat\ X\ Y,$
$eq\ (rpair\ Ra\ Rb)\ (pair\ X_1\ Y_1)\ (pair\ X_2\ Y_2) \to (eq\ Ra\ X_1\ X_2) \wedge (eq\ Rb\ Y_1\ Y_2)\ \}$

b) Equality using GADTs

Fig. 4. Type-indexed equality

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

We stress also the fact that the program of Fig. 4-a would be rejected by systems supporting GADTs [3,25], while the encoding of equality using GADTs as type representations in Fig. 4-b is also accepted by our type system.

Another interesting point is that we can handle equality in a quite fine way, much more flexible than in TOY or Curry, where equality is a *built-in* that proceeds structurally as in Fig. 4-a. With our proposed type system programmers can define structural equality as in Fig. 4-a for some types, choose another behavior for others, and omitting the rules for the cases they do not want to handle. Moreover, the type system protects against unsafe definitions, as we explain now: it is known [8] that in the presence of HO-patterns[2] structural equality can lead to the problem of *opaque decomposition*. For example, consider the expression $eq\ (snd\ z)\ (snd\ true)$. It is well-typed, but after a decomposition step using the structural equality we obtain $eq\ z\ true$, which is ill-typed. Different solutions have been proposed [8], but all of them need fully type-annotated expressions at run time, which penalizes efficiency. With the proposed type system that overloading at run time is not necessary since this problem of opaque decomposition is handled statically at compile time: we simply cannot write equality rules leading to opaque decomposition, because they are rejected by the type system. This happens with the rule $eq\ (snd\ X)\ (snd\ Y) \to eq\ X\ Y$, which will produce the previous problematic step. It is rejected because the inferred type for the right-hand side and its variables X and Y is $(bool, \gamma, \gamma)$, which is more specific than the inferred in the left-hand side $(bool, \alpha, \beta)$.

5.2 Existential Types, Opacity and HO Patterns

Existential types [14] appear when type variables in the type of a constructor do not occur in the final type. For example the constructor $key : \forall\alpha.\alpha \to (\alpha \to nat) \to key$ has an existential type, since α does not appear in the final type key. In functional logic languages, however, HO-patterns can introduce the same

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

opacity as constructors with existential type. A prototypical example is *snd X*: we know that X has some type, but we cannot know anything about it from the type $\beta \to \beta$ of the expression. In [17] a type system managing the opacity of HO-patterns is proposed. The program below shows how the system presented here generalizes [17], accepting functions that were rejected there (e.g. *idSnd*) and also supporting constructors with existential type (e.g. $getKey$):

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

Another remarkable example is given by the well-known translation of higher-order programs to first-order programs often used as a stage of the compilation of functional logic programs (see e.g. [18,1]). In short, this translation introduces a new function symbol @ ('*apply*'), adds calls to @ in some points in the program and appropriate rules for evaluating it. This latter aspect is interesting here, since the rules are not Damas-Milner typeable. The following program contains the @-rules (written in infix notation) for a concrete example with the constructors *z*, *s*, *nil*, *cons* and the functions *length*, *append* and *snd* with the usual types.

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

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

5.3 Generic Functions

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

This can be done by using GADTs to represent uniformly the applicative structure of expressions (for instance, the *spines* of [12]), by defining *size* over that uniform representations, and then applying it to concrete types via conversion functions. Again, we can also offer a similar but simpler alternative. A uniform representation of constructed data can be achieved with a data type

data univ = c nat [univ] where the first argument of c is for numbering constructors, and the second one is the list of arguments of a constructor application. A universal *size* can be defined as *usize (c _ Xs) → s (sum (map usize Xs))* using some functions of Haskell's prelude. Now, a generic *size* can be defined as $size \to usize \cdot toU$, where toU is a conversion function with declared type $toU : \forall\alpha.\alpha \to univ$

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

(s^i abbreviates iterated s's). This toU function uses the specific features of our system. It is interesting also to remark that in our system the truly generic rule $size \to usize \cdot toU$ can coexist with the type-indexed rules for *size* of Sect. 1. This might be useful in practice: one can give specific, more efficient definitions for some concrete types, and a generic default case via toU conversion for other types[3].

Admittedly, the type *univ* has less representation power than the spines of [12], which could be a better option in more complex situations. Nevertheless, notice that 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 [20] for some known problems for the latter).

Liberality (continued from Sect. 4). we recall the example of *size*, where our system accepts as well-typed *(size e)* for any well-typed e. Type classes impose more control: *size e* is only accepted if e has a type in the class *Sizeable*. There is a burden here: you need a class for each generic function, or at least for each range of types for which a generic function exists; therefore, the number of class instance declarations for a given type can be very high. GADTs are in the middle way. At a first sight, it seems that the types to which *size* can be applied are perfectly controlled because only *representable* types are permitted. The problem, as with classes, comes when considering other functions that are generic but for other ranges of types. Now, there are two options: either you enlarge the family of representable functions, facing up again the possibility of applying *size* to unwanted arguments, or you introduce a new family of representation types, which is a programming overhead, somehow against genericity.

[3] 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.

Need of type declarations. In contrast to Damas & Milner system, where principal types exist and can be inferred, our definition of well-typed program (Def. 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 [3,25]. 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 [27] one of the most remarkable applications of type systems was developed. The main idea there is to derive "free theorems" about the equivalence of functional expressions by just using the types of some of its constituent functions. These equivalences express different distribution properties, based on Reynold's abstraction theorem there recasted as "the parametricity theorem", which basically exploits the fact that a function cannot inspect the values of argument subexpressions with a polymorphic variable as type. Parametricity was originally developed for the polymorphic λ-calculus, so free theorems have to be weakened with additional conditions in order to accomodate them to practical languages like Haskell, as their original formulations are false in the presence of unbounded recursion, partial functions or impure features like seq [27,13].

With our type system parametricity is lost, because functions are allowed to inspect any argument subexpression, as seen in the *size* function from page 81. This has a limited impact in the FLP setting, since it is known that nondeterminism and narrowing—not treated in the present work but standard in FLP systems—not only breaks free theorems but also equational rules for concrete functions that hold for Haskell, like $(filter\ p) \circ (map\ h) \equiv (map\ h) \circ (filter\ (p \circ h))$ [4].

7 Conclusions

Starting from a simple type system, essentially Damas-Milners's one, we have proposed a new notion of well-typed functional logic program that exhibits interesting properties: simplicity; enough expressivity to achieve existential types or GADT-like encodings, and to open new possibilities to genericity; good formal properties (type soundness, protection against unsafe use of HO patterns). Regarding the practical interest of our work, we stress the fact that no existing FLP system supports any of the examples in Sect. 5, in particular the examples of the *equality*—where known problems of *opaque decomposition* [8] can be addressed—and *apply* functions, which play important roles in the FLP setting. Moreover, our work greatly improves our previous results [17] about safe uses of HO patterns. However, considering also the weaknesses discussed in Sect. 6 suggests that a good option in practice could be a partial adoption of our system,

not attempting to replace standard type inference, type classes or GADTs, but rather complementing them.

We find suggestive to think of the following future scenario for our system TOY: a typical program will use standard type inference except for some concrete definitions where it is annotated that our new liberal system is adopted instead. In addition, adding type classes to the languages is highly desirable; then the programmer can choose the feature—ordinary types, classes, GADTs or our more direct generic functions—that best fits his needs of genericity and/or control in each specific situation. We have some preliminary work [21] exploring the use of our type-indexed functions to implement type classes in FLP, with some advantages over the classical dictionary-based technology.

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

- A precise specification of how to mix different typing conditions in the same program and how to translate type classes into our generic functions.
- Despite of the lack of principal types, some work on type inference can be done, in the spirit of [25].
- Combining our genericity with the existence of modules could require adopting *open* types and functions [15].
- Narrowing, which poses specific problems to types, should be also considered.

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

References

1. Antoy, S., Tolmach, A.P.: Typed higher-order narrowing without higher-order strategies. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 335–353. Springer, Heidelberg (1999)
2. van Bakel, S., Fernández, M.: Normalization Results for Typeable Rewrite Systems. Information and Computation 133(2), 73–116 (1997)
3. Cheney, J., Hinze, R.: First-class phantom types. Tech. Rep. TR2003-1901, Cornell University (2003)
4. Christiansen, J., Seidel, D., Voigtländer, J.: Free theorems for functional logic programs. In: Proc. PLPV 2010, pp. 39–48. ACM, New York (2010)
5. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proc. POPL 1982, pp. 207–212. ACM, New York (1982)
6. González-Moreno, J.C., Hortalá-González, T., López-Fraguas, F., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. Journal of Logic Programming 40(1), 47–87 (1999)
7. González-Moreno, J., Hortalá-González, M., Rodríguez-Artalejo, M.: A higher order rewriting logic for functional logic programming. In: Hill, P.M., Warren, D.S. (eds.) Proc. ICLP 1997, pp. 153–167. MIT Press, Cambridge (1997)
8. Gonzalez-Moreno, J.C., Hortala-Gonzalez, M.T., Rodriguez-Artalejo, M.: Polymorphic types in functional logic programming. Journal of Functional and Logic Programming 2001(1) (2001)

9. Hanus, M.: Multi-paradigm declarative languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
10. Hanus, M. (ed.): Curry: An integrated functional logic language, version 0.8.2 (2006), http://www.informatik.uni-kiel.de/~curry/report.html
11. Hinze, R.: Generics for the masses. J. Funct. Program. 16(4-5), 451–483 (2006)
12. Hinze, R., Löh, A.: Generic programming, now! In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719, pp. 150–208. Springer, Heidelberg (2007)
13. Hudak, P., Hughes, J., Jones, S.P., Wadler, P.: A History of Haskell: being lazy with class. In: Proc. HOPL III, pp. 12-1–12-55. ACM, New York (2007)
14. Läufer, K., Odersky, M.: Polymorphic type inference and abstract data types. ACM Transactions on Programming Languages and Systems 16. ACM (1994)
15. Löh, A., Hinze, R.: Open data types and open functions. In: Proc. PPDP 2006, pp. 133–144. ACM, New York (2006)
16. López-Fraguas, F.J., Martin-Martin, E., Rodríguez-Hortalá, J.: Liberal Typing for Functional Logic Programs (long version). Tech. Rep. SIC-UCM, Universidad Complutense de Madrid (August 2010), http://gpd.sip.ucm.es/enrique/publications/liberalTypingFLP/long.pdf
17. López-Fraguas, F.J., Martin-Martin, E., Rodríguez-Hortalá, J.: New results on type systems for functional logic programming. In: Escobar, S. (ed.) Functional and Constraint Logic Programming. LNCS, vol. 5979, pp. 128–144. Springer, Heidelberg (2010)
18. López-Fraguas, F., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: Rewriting and call-time choice: the HO case. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 147–162. Springer, Heidelberg (2008)
19. López-Fraguas, F., Sánchez-Hernández, J.: $\mathcal{TOY}$: A multiparadigm declarative system. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
20. Lux, W.: Adding haskell-style overloading to curry. In: Workshop of Working Group 2.1.4 of the German Computing Science Association GI, pp. 67–76 (2008)
21. Martin-Martin, E.: Implementing type classes using type-indexed functions. To appear in TPF 2010 (2010), http://gpd.sip.ucm.es/enrique/publications/implementingTypeClasses/implementingTypeClasses.pdf
22. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348–375 (1978)
23. Moreno-Navarro, J.J., Mariño, J., del Pozo-Pietro, A., Herranz-Nieva, Á., García-Martín, J.: Adding type classes to functional-logic languages. In: Proc. APPIA-GULP-PRODE 1996, pp. 427–438 (1996)
24. Peyton Jones, S., Vytiniotis, D., Weirich, S.: Simple unification-based type inference for GADTs. Tech. Rep. MS-CIS-05-22, Univ. Pennsylvania (2006)
25. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proc. ICFP 2009, pp. 341–352. ACM, New York (2009)
26. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proc. POPL 1989, pp. 60–76. ACM, New York (1989)
27. Wadler, P.: Theorems for free! In: Proc. FPCA 1989, pp. 347–359. ACM, New York (1989)
28. Wright, A.K., Felleisen, M.: A Syntactic Approach to Type Soundness. Information and Computation 115, 38–94 (1992)