Generic Functional Logic Programs*

Technical Report SIC-03-10, 2010

Francisco J. López-Fraguas, Enrique Martin-Martin, Juan Rodríguez-Hortalá
Dpto. de Sistemas Informáticos y Computación
Facultad de Informática, Universidad Complutense de Madrid
fraguas@sip.ucm.es, emartinm@fdi.ucm.es, juanrh@fdi.ucm.es

ABSTRACT

In this paper we provide functional logic programs with generic programming capabilities. This is achieved by a remarkably simple extension of Damas-Milner type system. The most important aspect is a new notion of well-typed programs, which is liberal enough as to allow generic programming, but also restrictive enough as to ensure *type safety* for reductions, as we formally prove. We give an extense collection of examples showing the possibilities of our proposal, with one distinguished case of study addressing how to use our generic functions to implement *type classes*. We show that our approach compares favorably with the traditional dictionary-based approach to type classes implementation. Not only we overcome a known problem of interference of dictionaries with the semantics of non-determinism adopted by standard functional-logic languages, but also the resulting translated code is simpler and exhibits better performance than the code using dictionaries. Although our proposal has been devised with functional logic programs in mind, and at the formal level (for instance, to prove subject reduction) we rely on operational calculus proposed for those languages, our approach is also applicable to a pure functional setting, as many of our examples (in particular, the type classes case study) show.

Keywords

Functional logic programming, generic functions, type-indexed functions, existential types, type classes

1. INTRODUCTION

Functional logic programming. Modern functional logic languages [11] like Curry [12] or Toy [22] have a strong resemblance to lazy functional languages like Haskell [30]. A remarkable difference is that functional logic programs (FLP, in short) can be non-confluent, giving raise to non-deterministic functions, for which a *call-time choice* semantics [9] is adopted.

In the HO-CRWL approach to FLP [7], followed by the Toy system, programs can use *HO-patterns* (essentially, partial applications of symbols to other patterns) in left hand sides of function definitions. This corresponds to an *intensional* view of functions, i.e., different descriptions of the same 'extensional' function can be distinguished by the semantics. This is not an exoticism; it is known [21] that extensionality is not a valid principle within the combination of HO, non-determinism and call-time choice. Still, some people do not like HO-patterns in left-hand sides, one of the reasons being that they cause some bad interferences with types. Some works [8, 20] have addressed that problem, and this paper does further contributions to overcome it.

All those aspects of FLP will play a role in the paper, and in Section 3 we use a formal setting according to that. However, most of the paper can be read from a 'standard' functional programming perspective —where it offers also hopefully interesting novelties— 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, e.g., for 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 [27, 23], where in particular a technical treatment of the type system is absent.

The term *generic programming* is not used in the literature in a unique sense. With it, we refer generically to any situation in which a program piece (maybe something as simple as a function name) 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 [33] were invented to deal with those situations. Some further developments of the idea of generic programming (e.g. [14]) are based on type classes, while others (e.g. [15]) have preferred to use simpler extensions of Damas-Milner system, such as GADTs [4, 32] (see also [6] for a survey on generic functional programming).

Our paper aims at providing also FLP with abilities for generic programming. Noticeably, our solution is applicable also in a pure FP setting. The following example should serve to illustrate the main points.

An introductory example. Consider a program that manipulates several constructor data types: Peano natural numbers given by the constructors z, s[1], booleans and poly-

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

[1]We follow syntactic conventions of some functional logic languages where function and constructor names are lowercased, and variables are uppercased.

morphic 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 [] → *s z* *size (X:Xs)* → *s (add (size X) (size Xs))*

assuming a natural definition for *add*. 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. A price to pay is that the running program is not the program above, but the result of adding hidden dictionaries. GADT-based solutions would add an explicit representation of types to the encoding of *size* converting it into a so-called *type-indexed* function. GADTs are also supported by our system (see Section 3.3 and 4.1), 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. Moreover, the well-typedness criterion requires only a quite simple additional check over usual type inference for expressions.

'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 subject reduction: 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. Definition 3.1 states that with precision, and allows us to prove type safety (subject reduction + progress) for our system.

Contributions. We can identify a few of them:

• A novel (even in the FP setting) notion of well-typed program that induces a simple and direct way of programming type-indexed and generic functions is presented in Section 3.1. It is based on simple calculi for what essentially is Damas-Milner type derivation and inference (for expressions, not for programs). The approach supports also existential types and GADTs.

• At the formal level, we prove that well-typed programs enjoy *subject reduction*, an essential property for a type system. By introducing *failure* rules (a side contribution of our work) to the formal operational calculus, we also are able to ensure the *progress* property of well-typed programs (Section 3.4).

• We give a significant collection of examples showing the interest of the proposal in Section 4. As distinguished case study, in Section 4.4 we show how to use our generic functions to implement *type classes* in a way that can compete in simplicity and performance —we give experimental data— with the traditional dictionary-based approach to type classes implementation. Moreover, it overcomes a known problem of interference of dictionaries with call-time choice semantics.

• Our well-typedness criterion goes far beyond the solutions given in previous works [8, 20] to type-unsoundness problems of the use of *HO-patterns* in function definitions. We can type equality, solving known problems of *opaque decomposition* [8] (Section 4.3) and, most remarkably, we can type the *apply* function appearing in the HO-to-FO translation used in standard FLP implementations (Section 4.2).

2. PRELIMINARIES

We assume a signature $\Sigma = DC \cup FS$, where DC and FS are two disjoint sets of *data constructor* and *function* symbols resp., all them with associated arity. We write DC^n (resp FS^n) for the set of constructor (function) symbols of arity n, and if a symbol h is in DC^n or FS^n we write $ar(h) = n$. We consider an special constructor $fail \in DC^0$ to represent pattern matching failure in programs. We also assume a denumerable set $\mathcal{DV}$ of *data variables* X. Figure 1 shows the syntax of *patterns* $\in Pat$ and *expressions* $\in Exp$. We split the set of patterns in two: *first order patterns* $FOPat \ni fot ::= X \mid c\ t_1 \ldots t_n$ 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 $n \geq ar(f)$. $fv(e)$ is the set of variables in e which are not bound by any let expression and is defined in the usual way (notice that since our let expressions do not support recursive definitions the bindings 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* $\mathcal{C}$ is an expression with exactly one hole. 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 $f\ \overline{t} \to e$ where the set of patterns $\overline{t}$ is linear (there is not repetition of variables) and $fv(e) \subseteq \bigcup_{t_i \in \overline{t}} var(t_i)$. 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 $ar(C) = n$. Figure 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 $s_i \in DC \cup FS \cup \mathcal{DV}$. We only consider *coherent* sets of assumptions wrt. CS:

Definition 2.1 (Coherence of $\mathcal{A}$ wrt. CS). *A set of assumptions $\mathcal{A}$ is coherent wrt. a set of constructor symbols CS if for every constructor symbol c in CS different from $fail$, $c \in CS^n$ implies $\mathcal{A}(c) = \forall\overline{\alpha}.\tau_1 \to \ldots \to \tau_n \to (C\ \tau'_1 \ldots \tau'_m)$ with $ar(C) = m$; and $\mathcal{A}(fail) = \forall\alpha.\alpha$.*

Therefore the assumptions for constructors are related with their arity. On the other hand the assumption for *fail*, together with the rules of the type system presented

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

Figure 1: Syntax of expressions and programs

in Section 3.2, allow *fail* to appear at any place of a well-typed expression. 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 will say σ is an *instance* of σ' if $\sigma = \sigma'\pi$ for some π. A simple type τ' is a *generic instance* of $\sigma = \forall \overline{\alpha_n}.\tau$ if $\tau' = \tau[\overline{\alpha_n/\tau_n}]$ for some $\overline{\tau_n}$, and we write it $\sigma \succ \tau'$. Finally, τ' is a *variant* of $\sigma \equiv \forall \overline{\alpha_n}.\tau$ ($\sigma \succ_{var} \tau'$) if $\tau' = \tau[\overline{\alpha_n/\beta_n}]$ and $\overline{\beta_n}$ are fresh type variables.

3. FORMAL SETUP

3.1 Semantics

As the operational semantics of our functional logic programs we have chosen *let*-rewriting[21], a simple and high level notion of reduction step devised to express the semantics of call-time choice, inspired by [2].We have extended it for this paper with two more rules for managing failure of pattern matching (Figure 2), an aspect that can be interesting in its own. The new rule (Ffail) generates a pattern matching failure when no program rule can be used to reduce an expression. 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 $\mathcal{P}_1 = \{true \wedge X \to X, false \wedge X \to false\}$ 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 $Y \wedge true$. Using unification in (Ffail) can contribute also to early detection of failures. Consider the program $\mathcal{P}_2 = \{f\ true\ false \to true, g \to g\}$ and the expression $let\ Y = g\ in\ f\ Y\ Y$. Since $f\ Y\ Y$ does not unify with $f\ true\ false$, (Ffail) detects a failure, while other operational approaches to failure in FLP [31] would lead to a divergence. Notice that, in presence of non-determinism, call-time choice semantics is essential for (Ffail) to be sound. Finally, rule (FailP) is used to propagate the pattern matching failure when it is applied to other expression.

Notice that since *let*-rewriting is lazy, the occurrence of a pattern matching failure in a subexpression does not always force the whole expression to be reduced to *fail*.

3.2 Type system and type inference

Both derivation and inference rules are based on those presented in [20, 25] Our type derivation rules for expressions (Figure 3-a) correspond to the well-known variation of Damas-Milner's [5] type system with syntax-directed rules. $Gen(\tau, \mathcal{A})$ 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: $Gen(\tau, \mathcal{A}) = \forall \overline{\alpha_n}.\tau\ where\ \{\overline{\alpha_n}\} = ftv(\tau) \setminus ftv(\mathcal{A})$.

The type inference algorithm $\Vdash$ (Figure 3-b) follows the same ideas that $\mathcal{W}$[5]. We have given the type inference a relational style to show the similarities with the typing rules. But in essence, the inference rules represent an algorithm which fails if any of the rules cannot 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 the "most general" substitution we have to apply to $\mathcal{A}$ in order to be able to derive any type for e.

3.3 Well-typed programs

In the functional setting, the problem of checking if a program is well-typed is reduced to checking that an expression has a valid type, because programs can be written as a chain of let expressions defining the functions accompanied with a goal expression to evaluate. Doing that in our formalism would not preserve call-time choice semantics and, furthermore, we do not consider λ-abstractions. Therefore we need an explicit definition of how to check that a program is well-typed. The following definition is the *main contribution* of this paper, and it is based on type inference for expressions:

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 $\mathcal{A}$ ($wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e)$) iff:*

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

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

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

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

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

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

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

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

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

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

Figure 2: Higher order *let*-rewriting relation with pattern matching failure $\rightarrow^{lf}$

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

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 $\begin{array}{l}\alpha\ fresh\ type\ variable \wedge \\ \pi = mgu(\tau_1\pi_2, \tau_2 \rightarrow \alpha)\end{array}$

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

b) Type inference rules

Figure 3: Type system

where $\{\overline{X_n}\} = var(f\ t_1 \ldots t_n)$ 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 type inferred for the right-hand side and the variables appearing in it are more general than the inferred for the left-hand side. This fact guarantees the *subject reduction* property (i.e., that an expression resulting after a reduction step can have the same type as the original one) when applying a program rule. Moreover, this point ensures a correct management of both *skolem* constructors [18] and *opaque* variables [20], introduced either by the presence of existentially qualified constructors or higher order patterns. Finally, the last point guarantees that the set of assumptions is not modified neither by the type inference nor the matching substitution.

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

$$\begin{array}{ll}\mathcal{A} \equiv \{ & \mathbf{z} : nat, \mathbf{s} : nat \rightarrow nat, \mathbf{true} : bool, \mathbf{false} : bool, \\ & \mathbf{cons} : \forall\alpha.\alpha \rightarrow [\alpha] \rightarrow [\alpha], \mathbf{rnat} : repr\ nat \\ & \mathbf{snd} : \forall\alpha, \beta.\alpha \rightarrow \beta \rightarrow \beta, \\ & \mathbf{cast} : \forall\alpha, \beta.(\alpha \rightarrow \alpha) \rightarrow \beta, \\ & \mathbf{eq} : \forall\alpha \rightarrow \alpha \rightarrow bool, \\ & \mathbf{show} : \forall\alpha.repr\ \alpha \rightarrow \alpha \rightarrow [char], \\ & \mathbf{showNat} : nat \rightarrow [char] \\ & \mathbf{force} : \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow [nat] \quad \}\end{array}$$

$$\begin{array}{ll}\mathcal{P} \equiv \{ & snd\ X\ Y \rightarrow Y, cast\ (snd\ X) \rightarrow X, \\ & eq\ (s\ X)\ z \rightarrow false, \\ & show\ rnat\ X \rightarrow showNat\ X, \\ & force\ (snd\ X) \rightarrow cons\ z\ X \quad \}\end{array}$$

The rule of snd *is trivially well-typed. The rule for* cast *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 which "escape" from the scope. If that rule were well-typed then subject reduction could not be granted anymore—e.g. consider the step $cast\ (snd\ true) \rightarrow^{lf} true$, where the type nat can be assigned to $cast\ (snd\ true)$ but true can only have type bool.*

The rule 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 the left-hand side and $([char], nat)$ for the right-hand side, so it is well-typed. Finally, the rule for* force *is not well-typed because the tuple $([nat], \gamma)$ in-*

ferred for the left-hand side is not an instance of the tuple $([nat],[nat])$ *inferred in the right-hand side. This shows the problem of a existential type variable (*rigid variable*) which has a concrete type in the right hand side.*

Notice that the last point of the definition holds trivially in all the rules because $\mathcal{A}$ *is closed.*

The examples of well-typed rules, together with the fact that well typedness for programs simply proceeds rule by rule, suggest the capabilities of our notion for type-indexed functions already described in Example 3.1 and further exploited in Section 4. The examples of ill-typed rules show a good control of existential types that could come either from HO patterns (like *snd X*) or the assumptions for constructor symbols.

The definition of well-typed rule is simple and can be implemented easily. The first two points use the algorithm of type inference for expressions, while the third is just a matching. The last point, which seems harder, requires to calculate the intersection between the domain of three different substitutions and the free type variables of a set of assumptions. However, as we will see in Section 5, this point is trivial when type-checking real programs because the assumptions will be closed.

The definition of well-typed rule assumes that a type assumption is given for each function, otherwise the inference of the left-hand side fails. This implies that this definition cannot be used as a type inference mechanism for the types of the functions, but as a type checking method. It is not a major weakness, since explicit type declarations are mandatory in the presence of polymorphic recursion [17] and other common extensions of the type system like arbitrary-rank polymorphism [28, 16] or GADTs [4, 32]. Moreover, programmers do include type annotations for functions in order to document programs. In any case, type checking for generic functions and usual type inference can be combined in real programs, as discussed in Section 5.

Another point making type inference for functions difficult is that functions, unlike expressions, do not have principal types. Consider the following rule:

$$not\ true \to false$$

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 more general than the others.

In [20] other definition of well-typed programs based on type derivations is proposed (we write $wt^{old}_{\mathcal{A}}(\mathcal{P})$ for that). We have proved that programs accepted there are also accepted in the type system proposed in this paper:

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

This result is specially relevant taking into account that the type system from [20] is equivalent to Damas-Milner's system for programs not containing higher order patterns. We remark that all the examples in Section 4 are rejected as ill-typed by the type system in [20]. Therefore, the type system proposed here is a strict and useful generalization.

3.4 Properties of the type system

Our type system is more general than the classical Damas-Milner type system or the type system in [20, 25]. However, it still has the needed properties demanded to type systems: subject reduction and progress. The following theorem shows that if an expression e with type τ is rewritten to other expression e' in one step, then e' has also type τ.

THEOREM 3.2 (SUBJECT REDUCTION). *If* $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ *and* $\mathcal{P} \vdash e \to^{lf} e'$, *then* $\mathcal{A} \vdash e' : \tau$.

Theorem 3.3 states that well-typed ground expressions are not "stuck" wrt. a well-typed program, i.e., they are patterns or we can perform a *let*-rewriting step.

THEOREM 3.3 (PROGRESS). *If* $wt_{\mathcal{A}}(\mathcal{P})$, *e is ground and* $\mathcal{A} \vdash e : \tau$ *then either e is a pattern or* $\exists e'.\ \mathcal{P} \vdash e \to^{lf} e'$.

The following result is a straightforward combination of the two previous theorems.

THEOREM 3.4 (TYPE SAFETY). *Assume* $wt_{\mathcal{A}}(\mathcal{P})$, *e is ground and* $\mathcal{A} \vdash e : \tau$. *Then one of the two following conditions holds:*

a) e is a pattern.

b) $E = \{e' | \mathcal{P} \vdash e \to^{lf} e'\} \neq \emptyset$ *and for every* $e' \in E$, $\mathcal{A} \vdash e' : \tau$.

4. EXAMPLES

In this section there are 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 an initial set of assumptions:

$$\begin{array}{ll} \mathcal{A}_{basic} \equiv \{ & \mathbf{true} : bool, \mathbf{false} : bool, \\ & \mathbf{z} : nat, \mathbf{s} : nat \to nat, \\ & \mathbf{pair} : \forall\alpha,\beta.\alpha \to \beta \to pair\ \alpha\ \beta \\ & \wedge, \vee : bool \to bool \to bool, \\ & \mathbf{snd} : \forall\alpha,\beta.\alpha \to \beta \to \beta, \\ & \mathbf{nil} : \forall\alpha.[\alpha], \mathbf{cons} : \forall\alpha.\alpha \to [\alpha] \to [\alpha] \quad \} \end{array}$$

4.1 Type-indexed functions

Type-indexed functions (in the sense appeared in [15]) are functions that have a particular definition for each type in a certain family. The function *size* of Section 1 was an example of such a function. A similar example is given in Figure 4-a, containing the code for an equality function which only operates with booleans, natural numbers and pairs.

An interesing 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[4, 15]. 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 two arguments of every rule already fix the type of the left-hand side and its variables to be more specific (or the same) than the inferred in the right-hand side. The absence of extra arguments 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 arguments. 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. [11]), but that cannot be written directly in existing systems like Curry or Toy, because

$\mathcal{A} \equiv \quad \mathcal{A}_{basic} \oplus \{\mathbf{eq} : \forall\alpha.\alpha \to \alpha \to bool\}$
$\mathcal{P} \equiv \{ \quad eq\ true\ true \to true,$
$eq\ true\ false \to false,$
$eq\ false\ true \to false,$
$eq\ false\ false \to true,$

$eq\ z\ z \to true,$
$eq\ z\ (s\ X) \to false,$
$eq\ (s\ X)\ z \to false,$
$eq\ (s\ X)\ (s\ Y) \to eq\ X\ Y,$

$eq\ (pair\ X_1\ Y_1)\ (pair\ X_2\ Y_2) \to$
$\quad (eq\ X_1\ X_2) \wedge (eq\ Y_1\ Y_2) \qquad \}$

a) Original program

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

$\mathcal{P} \equiv \{ \quad eq\ rbool\ true\ true \to true,$
$eq\ rbool\ true\ false \to false,$
$eq\ rbool\ false\ true \to false,$
$eq\ rbool\ false\ false \to true,$

$eq\ rnat\ z\ z \to true,$
$eq\ rnat\ z\ (s\ X) \to false,$
$eq\ rnat\ (s\ X)\ z \to false,$
$eq\ rnat\ (s\ X)\ (s\ Y) \to eq\ rnat\ X\ Y,$

$eq\ (rpair\ Ra\ Rb)\ (pair\ X_1\ Y_1)\ (pair\ X_2\ Y_2) \to$
$\quad (eq\ Ra\ X_1\ X_2) \wedge (eq\ Rb\ Y_1\ Y_2) \qquad \}$

b) Equality using GADTs

Figure 4: Type-indexed equality

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 [4, 32], while the encoding of equality using GADTs as type representations in Figure 4-b) is accepted by our type system.

Another interesting point is that we can handle equality in a quite fine way, much more flexible than in Curry or Toy, 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 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 *higher order patterns*[2] structural equality can lead to the problem of *opaque decomposition*. For example, consider the **snd** function defined by the rule $snd\ X\ Y \to Y$ and with type $\forall\alpha,\beta.\alpha \to \beta \to \beta$. In this situation $eq\ (snd\ z)\ (snd\ true)$ 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 our 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 more specific than the inferred in the left-hand side $(bool, \alpha, \beta)$. This is a good example of how our system combines flexibility with control. Our next section pursues that idea.

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

4.2 Existential types, opacity and HO patterns

Existential types [26, 18] 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*. This type is called existential because it can be viewed as $(\exists\alpha.\alpha \to (\alpha \to nat)) \to key$. In practice, this means that a *key* constructor contains a value of *some* type and a function from *that* type to natural numbers, but we cannot know exactly what type it is.

In functional logic languages, higher order patterns can introduce the same opacity than existentially quantified constructors. A prototypical example is $snd\ X$. Here we know that X has some type, but we cannot know anything about it from the type $\beta \to \beta$ of the expression. A type system managing the opacity appearing in higher order patterns is proposed in [20]. In that paper a distinction is made between *opaque* and *transparent* variables. Then, only transparent variables (those whose type is univocally fixed by the type of the pattern that contains them) are allowed in right-hand sides. Therefore a rule as $g\ (snd\ X) \to snd\ X$ is rejected, since X is opaque in the pattern $snd\ X$ and appears in the right hand side. The system we propose in this paper generalizes [20], accepting functions which were rejected by the type system of those papers. As an example, the previous rule for g is well-typed wrt. the assumption $g : \forall\alpha,\beta.(\alpha \to \alpha) \to (\beta \to \beta)$. This is desirable, since the rule is the identity over $snd\ X$ and it will not generate any error at run time despite the opacity.

Figure 5 shows a stack, a typical example using existential types borrowed from [18]. The constructor *stack* has an existential type because the type of its first argument does not appear in the final type. This first argument is the concrete implementation of the stack, and the other arguments are functions that push an element in the stack, remove the last element and ask for the last element. We do not know anything about the implementation of the stack, but we are sure that the contained functions operate correctly with that implementation. Therefore we can write the functions *push*, *pop* and *top*, which extract the respective function and apply it to the implementation. We can create stacks with different implementations, but these functions will work for all the stacks. Notice that all the rules (except *makeListStack*) are ill-typed in [20]. This example shows how using existential types we can create first-class abstract data types that can be passed and returned by functions. Therefore at runtime we can mix stacks with different implementations, crate copies of stacks with different implementations or add new implementations. As far as we know, this kind of flexibility cannot be easily achieved using modules.

Another remarkable example is given by the well-known translation to first order usually used as a stage of the com-

stack : $\forall\alpha,\beta.\beta \to (\alpha \to \beta \to \beta) \to$ $(\beta \to \beta) \to (\beta \to \alpha) \to stack\ \alpha$	$push\ X\ (stack\ R\ Push\ Pop\ Top) \to stack\ (Push\ X\ R)\ Push\ Pop\ Top$
push : $\forall\alpha.\alpha \to stack\ \alpha \to stack\ \alpha$	$pop\ (stack\ R\ Push\ Pop\ Top) \to stack\ (Pop\ R)\ Push\ Pop\ Top$
pop : $\forall\alpha.stack\ \alpha \to stack\ \alpha$	$top\ (stack\ R\ Push\ Pop\ Top) \to Top\ R$
top : $\forall\alpha.stack\ \alpha \to \alpha$	$makeListStack \to stack\ nil\ cons\ tail\ head$
makeListStack : $\forall\alpha.stack\ \alpha$	

Figure 5: Module stack as a first-class citizen using existential types

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

Figure 6: Translation HO-to-FO

pilation of functional logic programs (see e.g. [21, 1]).

In short, this translation introduces a new function symbol @, adds calls to @ in some points in the program, and adds appropriate rules for evaluating it. It is this latter aspect which is interesting here, since the rules are not Damas-Milner typeable. Fig. 6 contains the @-rules for a concrete example involving the data constructors *z*, *s*, *nil*, *cons* and the functions *length*, *append* and *snd* with the usual types. 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 Definition 3.1 they are all well-typed, provided we declare @ to have the type $@ : \forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta$. As a consequence 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.

4.3 Generic functions

According to a strict view of genericity, the functions *size* and *eq* in Sections 1 and 4.1 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 [15]), by defining *size* over that uniform representations, and then applying it to concrete types via conversion functions. But again, we can also offer a similar but simpler alternative. A uniform, universal representation of constructed data can be achieved with a standard data type

$$data\ univ = c\ nat\ [univ]$$

where the first argument of c is for numbering constructors, and the second is the list of arguments of a constructor application. A universal *size* can be then defined as:

$$usize\ (c\ \_\ Xs) \to s\ (foldr\ add\ z\ (map\ usize\ Xs))$$

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). It is in this toU function where we use 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 [3]

Admittedly, the type *univ* has less representation power than the spines of [15], that could be a better option in more complex situations. But notice that since spines are based on GADTs, they are also supported by us.

4.4 Type Classes

Type classes [33, 10] are, according to some authors, 'the most beloved feature of Haskell'. They provide a clean, modular and elegant way of writing overloaded functions. Type classes are usually implemented by means of dictionary passing: every overloaded function receives a dictionary at run time which contains the concrete function to execute. In this section we expose a new translation of type classes using type-indexed functions and type witnesses. This translation is interesting for a number of reasons. First, it obtains a gain in efficiency over the usual translation using dictionaries [33, 10], fact that we have measured experimentally. Second, it also solves a problem of excess of sharing that appears in functional logic languages when using dictionaries: the evaluation of expressions in translated programs may lose expected values. Finally the translation is simpler, which makes the implementation easier and also results in smaller and more understable translated programs.

4.4.1 Translation to type-indexed functions

The classical translation of type classes using dictionaries [33, 10] has been used for years in functional systems like Haskell, and several optimizations have been developed [3,

[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.

29]. Here we show a different translation that takes advantage of the flexibility provided by our type system and uses type-indexed functions and type witnesees instead of dictionaries. Consider the following simple program for searching in a sorted list. Notice that we have used Haskell syntax for the original program with type classes, and our syntax for the translated programs.

```
class Eq a where
  eq :: a -> a -> Bool

class Eq a => Ord a where
  less :: a -> a -> Bool

data Nat = Z | S Nat

instance Eq Nat where
  eq Z Z         = True
  eq Z (S x)     = False
  eq (S x) Z     = False
  eq (S x) (S y) = eq x y

instance Ord Nat where
  less Z Z         = False
  less Z (S x)     = True
  less (S x) Z     = False
  less (S x) (S y) = less x y

search :: Ord a => a -> [a] -> Bool
search x []     = False
search x (y:ys) = (eq x y) ||
  (less y x && search x ys)
```

There are only two type classes: `Eq` and `Ord`, being `Ord` a subclass of `Eq`. Both classes `Eq` and `Ord` contain only one function, `eq` and `less` respectively. The function `search` takes an element (in the `Ord` class) and a sorted list, and returns whether the given element is in the list or not using the overloaded functions `eq` and `less`. We assume that an usual type inference algorithm supporting type-classes [33, 10] has inferred the types of functions and expressions in the program, types that we will use in both translations.

Fig. 7-a) shows the usual translation of this program using dictionaries. First, every class declaration generates a dictionary type (*dictEq* and *dictOrd*) containing the member functions and the dictionaries of its direct superclasses. Every member function (*eq* and *less*) is converted into a projecting function which extracts the concrete implementation from the dictionary. Functions to extract superclass dictionaries from subclass dictionaries (*getEqfromOrd*) are also needed. Every instance generates concrete implementations of the member functions (*eqNat* and *lessNat*) with the suitable type, as well as particular dictionaries (*dictEqNat* and *dictOrdNat*) containing that functions and the concrete dictionaries of its superclasses. Finally, every function (*search*) is extended appending as many dictionaries as class restrictions appears in the context of its type and placing that dictionaries as arguments of the overloaded functions.

The alternative translation using type-indexed functions and type witnesses appears in Figure 7-b). This translation is simpler, generating a program approximately as long as the original one. The steps are:

1. Each data type is extended with a *type witness* for that type. The type witness of the data type $C\ \overline{\alpha_n}$ is $\#C$ with type $\forall\overline{\alpha_n}.\overline{\alpha_n} \to C\ \overline{\alpha_n}$. In the example program, the witness generated for *nat* is simply $\#nat$ of type *nat*. We use the special symbol # to be sure that the constructor $\#C$ cannot appear in the program, and to provide homogeneity to the names of the type witnesses. Type witnesses are a way of representing types as values which follows the same idea of *type representations* [4, 15]. For example, type witnesses of [*bool*] and *pair bool nat* would be $\#list\ \#bool$ and $\#pair\ \#bool\ \#nat$ respectively.

2. Member functions are extended with an extra argument representing the type witness. In the example, both *eq* and *less* functions of types $\forall\alpha.Eq\ \alpha \Rightarrow \alpha \to \alpha \to bool$ and $\forall\alpha.Ord\ \alpha \Rightarrow \alpha \to \alpha \to bool$ are extended to $\forall\alpha.\alpha \to \alpha \to \alpha \to bool$. The new first argument is the type witness of the elements to compare. Like in the translation using dictionaries, each occurrence of an overloaded function in the right-hand side must be completed with the corresponding type witness. In the example, the rule `eq (S x) (S y) = eq x y` is extended to $eq\ \#nat\ (s\ X)\ (s\ Y) \to eq\ \#nat\ X\ Y$.

3. Normal functions using overloaded functions in its right-hand side must also be completed. For every type with a class constraint in the context of the type, we need to add a type witness as an argument. Besides, it is also necessary to place the type witness as arguments of the overloaded functions. As an example, the original rule

```
search x (y:ys) = (eq x y) ||
  (less y x && search x ys)
```

is extended to

$$search\ Witness\ X\ (cons\ Y\ Ys) \to (eq\ Witness\ X\ Y) \vee (less\ Witness\ Y\ X \wedge search\ Witness\ X\ Ys)$$

Since the type of `search` is `Ord a => a -> [a] -> Bool` we need to add a type witness as the first argument *Witness*. Then this type witness is passed to the overloaded functions, namely *eq*, *less* and *search*.

4. As in the translation using dictionaries, goals are extended introducing the concrete type witnesses. For example the goal `search Z [S Z, S Z]` is translated to $search\ \#nat\ z\ [s\ z, s\ z]$.

Although in Section 4.1 we show how our type system allows writing type-indexed functions without extra parameters, in the translation of type classes we have included them. The reason is that type witness are sometimes necessary: when the type variable with the class constraint appears only on the result type, and when the rules do not use constructors in the left-hand side. As an example of the first situation consider the following program:

```
class Gen a where
  gen :: Bool -> a

instance Gen Nat where
  gen False = Z
  gen True  = S Z
```

If we do not use a type witness, the translated rules of `gen` in the instance `Gen Nat` are $gen\ false \to z$ and $gen\ true \to s\ z$. They are ill-typed wrt. the type $gen : \forall\alpha.bool \to \alpha$, because the type inferred for the right-hand side (*nat*) is more specific than the type inferred for the left-hand side (α). When we introduce the type witness $\#nat$ to the rules we obtain $gen\ \#nat\ false \to z$ and $gen\ \#nat\ true \to s\ z$, and the type $gen : \forall\alpha.\alpha \to bool \to \alpha$. Here, type witness forces both sides to have type *nat*, making the rules well-typed.

$\mathcal{A}_a \equiv \mathcal{A}_{basic} \oplus \{\mathbf{dictEq} : \forall\alpha.(\alpha \to \alpha \to bool) \to dictEq\ \alpha, \mathbf{dictOrd} : \forall\alpha.dictEq\ \alpha \to (\alpha \to \alpha \to bool) \to dictOrd\ \alpha,$
$\mathbf{eq} : \forall\alpha.dictEq\ \alpha \to (\alpha \to \alpha \to bool), \mathbf{less} : \forall\alpha.dictOrd\ \alpha \to (\alpha \to \alpha \to bool), \mathbf{getEqfromOrd} : \forall\alpha.dictOrd\ \alpha \to dictEq\ \alpha,$
$\mathbf{eqNat} : nat \to nat \to bool, \mathbf{lessNat} : nat \to nat \to bool,$
$\mathbf{dictEqNat} : dictEq\ nat, \mathbf{dictOrdNat} : dictOrd\ nat, \mathbf{search} : \forall\alpha.dictOrd\ \alpha \to \alpha \to [\alpha] \to bool\}$

$\mathcal{P}_a \equiv \{eq\ (dictEq\ X) \to X, less\ (dictOrd\ X\ Y) \to Y, getEqFromOrd\ (dictOrd\ X\ Y) \to X,$
$eqNat\ z\ z \to true, eqNat\ z\ (s\ X) \to false, eqNat\ (s\ X)\ z \to false, eqNat\ (s\ X)\ (s\ Y) \to eqNat\ X\ Y,$
$lessNat\ z\ z \to false, lessNat\ z\ (s\ X) \to true, lessNat\ (s\ X)\ z \to false, lessNat\ (s\ X)\ (s\ Y) \to lessNat\ X\ Y,$
$dictEqNat \to dictEq\ eqNat, dictOrdNat \to dictOrd\ dictEqNat\ lessNat,$
$search\ Dict\ X\ nil \to false,$
$search\ Dict\ X\ (cons\ Y\ Ys) \to (eq\ (getEqFromOrd\ Dict)\ X\ Y) \vee (less\ Dict\ Y\ X \wedge search\ Dict\ X\ Ys)\}$

$\mathcal{A}_b \equiv \mathcal{A}_{basic} \oplus \{\mathbf{eq} : \forall\alpha.\alpha \to \alpha \to \alpha \to bool, \mathbf{less} : \forall\alpha.\alpha \to \alpha \to \alpha \to bool, \#\mathbf{nat} : nat, \mathbf{search} : \forall\alpha.\alpha \to \alpha \to [\alpha] \to bool\}$

$\mathcal{P}_b \equiv \{eq\ \#nat\ z\ z \to true, eq\ \#nat\ z\ (s\ X) \to false, eq\ \#nat\ (s\ X)\ z \to false, eq\ \#nat\ (s\ X)\ (s\ Y) \to eq\ \#nat\ X\ Y,$
$less\ \#nat\ z \to false, less\ \#nat\ (s\ X) \to true, less\ \#nat\ (s\ X)\ z \to false, less\ \#nat\ (s\ X)\ (s\ Y) \to less\ \#nat\ X\ Y,$
$search\ Witness\ X\ nil \to false,$
$search\ Witness\ X\ (cons\ Y\ Ys) \to (eq\ Witness\ X\ Y) \vee (less\ Witness\ Y\ X\ \wedge\ search\ Witness\ X\ Ys)\}$

Figure 7: Translation using dictionaries $(\mathcal{A}_a, \mathcal{P}_a)$, and with type-indexed functions and type witnesses $(\mathcal{A}_b, \mathcal{P}_b)$

As an example of the second situation consider the translation in Figure 7-b). It is well-typed if we drop the type witnesses, since the constructors in the left-hand side force the variables to have a more specific type than in the right-hand side. For example in the rule $eq\ (s\ X)\ (s\ Y) \to eq\ X\ Y$, the variables X and Y have type nat in the left-hand side (because of the s constructor) but they are inferred a type α (a fresh type variable) in the right-hand side. But the programmer could have written the instance of `Eq` in a slightly different way:

```
instance Eq Nat where
  eq = eqNat
```

being `eqNat` a function that calculates the equality of naturals. The resulting program without type witnesses would be rejected because the rule $eq \to eqNat$ is ill-typed wrt. the assumption $eq : \forall\alpha.\alpha \to \alpha \to bool$ as the right-hand side has a more specific type ($nat \to nat \to bool$) than the left-hand side ($\alpha \to \alpha \to bool$). Using a type witness the rule will be $eq\ \#nat \to eqNat$, which now is well-typed since both sides have type $nat \to nat \to bool$ wrt. the assumption $eq : \forall\alpha.\alpha \to \alpha \to \alpha \to bool$.

The possibility of a translation using type representations instead of dictionaries was already mentioned in [4], although it was not further developed. In the next two subsections we will show that this translation generates more efficient programs and also solves a problem of excess of sharing in FLP when using the common translation of dictionaries.

4.4.2 *Efficiency*

The structure of type witnesses depend on the data type, unlike dictionaries, whose structure depends on the type class: they contain the member functions and the dictionaries of the direct superclasses. It is difficult to compare their size because it is highly program dependent. Programs with many nested type classes and many member function will require big dictionaries even for basic data, while programs with few classes and member functions will need big type witnesses if the overloaded functions are applied to complex data.

With dictionaries, member functions must be extracted before applying them, and it can mean traversing a chain of superclasses dictionaries [4]. Another point in favor is that we need less type witnesses than dictionaries. Consider a function with type `f :: (Eq a, Show a) => a -> Bool`. In the common translation, we introduce a dictionary for every class constraint in the type of the function. Therefore this function needs two extra paremeteres: $f : \forall\alpha.dictEq\ \alpha \to dictShow\ \alpha \to \alpha \to bool$. However, only one type witness is necessary, since both class contraints `Eq a` and `Show a` affect the same type `a`: $f : \forall\alpha.\alpha \to \alpha \to bool$.

We have measured the real gain in efficiency obtained using type witnesses and type-indexed functions instead of handwritten dictionaries and built-in type classes over different systems. Translated programs using dictionaries are valid in all the systems since they only need a Damas-Milner type system. Programs using type witnesses and type-indexed functions require an implementation with our proposed type system. None of the tested systems support it, so we have followed two solutions. For TOY 2.3.1 we have used an special version disabling type checking. This does not distort the measures since once compiled, programs do not carry any type information at run time, so compiled programs are the same regardless of the chosen type system. For the rest of the systems (GHC 6.10.4, Hugs Sept. 2006 and PAKCS 1.9.1(7) [13]) we have included all the data constructors inside the same universal data type. Thus, only one instance for each type class is needed and all the rules fit inside, making the program well-typed.

We have measured the speedup in programs with different levels of subclasses (Figure 8). The obtained speedup grows almost linearly with the subclass depth in all the cases except in GHC using handwritten dictionaries. In general terms the translation using type witnesses performs faster, with an speedup ranging from 1–1.3 using one subclass to 1.5–2.5 when nine subclasses are used. The only exception is in GHCi using the built-in type classes, where the translation using type witnesses runs about a 10% slower for one subclass, although for higher subclass depth the time

[4] Although this problem can be solved flattening the dictionary structure [3]

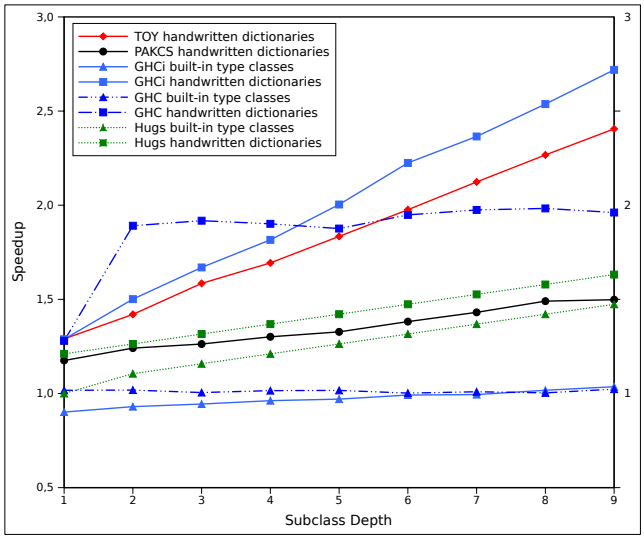

Figure 8: Speedup of the translation over different systems

used in both translation is the same. Detailed results and source programs can be found in `http://gpd.sip.ucm.es/enrique/publications/genericFLP/speedup.zip`.

As Figure 8 shows, in all the Haskell systems the obtained speedup is greater when compared to handwritten dictionaries than when compared to built-in type classes. This motivates us to think that these systems do not use the usual dictionary-based translation but a more sophisticated, optimized one. However, our proposed translation using type witnesses can compete in efficiency with this optimized translation (being the only exception GHCi when few subclasses are used). On the other hand, the translation using type witnesses obtains an interesting gain in efficiency compared to handwritten dictionaries, where the translation was performed manually by us following the specifications from [33, 10]. These facts encourage us to use type witnesses instead of dictionaries when implementing type classes in FLP languages, since the obtained efficiency is comparable or even better to the one obtained with the dictionary technique in either cases. Moreover, our proposed translation solves a particular problem of type classes in FLP, as we will see in the next section.

4.4.3 Problems with dictionaries in FLP

Apart from a gain in efficiency, the translation using type witnesses and type-indexed functions also solves a problem of undesired sharing that appears with type classes and non-determinism if dictionaries are used. This problem, which affects type classes with member functions without arguments, was pointed out in [24], as we explain now. Consider the following code:

```
class Arb a where          instance Arb Bool where
  arb :: a                   arb = True
                             arb = False

arbP2 :: (Arb a, Arb b) => (a,b)
arbP2 = (arb,arb)

arbL2 :: Arb a => [a]
arbL2 = [arb,arb]
```

The expression `arbP2::(Bool,Bool)` is translated into $arbP2\ dictArbBool\ dictArbBool$ whose evaluation generates precisely the values $(true, true)$, $(true, false)$, $(false, true)$, and $(false, false)$. However, the expression `arbL2::[Bool]` is translated into $arbL2\ dictArbBool$ which generates only the values $[true, true]$ and $[false, false]$. The reason is that *call-time choice* semantics imposes that the arguments of a function must be evaluated to a pattern before applying them. The following reduction using *let*-rewriting shows the problem:

$$
\begin{array}{ll}
 & \underline{arbL2\ dictArbBool} \\
\to^{lf}_{(LetIn)} & let\ D = dictArbBool\ in\ \underline{arbL2\ D} \\
\to^{lf}_{(Fapp)} & let\ D = \underline{dictArbBool}\ in\ [arb\ D, arb\ D] \\
\to^{lf}_{(Fapp)} & \underline{let\ D = dictArb\ arbBool\ in\ [arb\ D, arb\ D]} \\
\to^{lf}_{(LetIn,Flat)} & let\ B = arbBool\ in \\
 & \quad \underline{let\ D = dictArb\ B\ in\ [arb\ D, arb\ D]} \\
\to^{lf}_{(Bind)} & let\ B = arbBool\ in \\
 & \quad [\underline{arb\ dictArb\ B}, \underline{arb\ dictArb\ B}] \\
\to^{lf}_{(Fapp,Fapp)} & let\ B = arbBool\ in\ [B, B]
\end{array}
$$

Notice that we cannot apply (Bind) with D bound to $dictArb\ arbBool$ because that is not a pattern, as $arbBool$ is a function totally applied. Hence we introduce a new binding for $arbBool$ so after some *let* flattening the (Bind) step becomes enabled, and we can continue evaluating the two copies of $arb\ dictArb\ B$. We end up with a *let* in which is obvious that the two elements of the resulting list will share its value.

Using type witnesses we solve this problem:

$$
\begin{array}{ll}
\mathcal{A} \equiv \{ & arb : \forall\alpha.\alpha \to \alpha, \\
 & arbP2 : \forall\alpha,\beta.\alpha \to \beta \to (\alpha,\beta), \\
 & arbL2 : \forall\alpha.\alpha \to [\alpha] \qquad \} \\
\mathcal{P} \equiv \{ & arb\ \#bool \to true, arb\ \#bool \to false, \\
 & arbP2\ WitA\ WitB \to (arb\ WitA, arb\ WitB), \\
 & arbL2\ Wit \to [arb\ Wit, arb\ Wit] \qquad \}
\end{array}
$$

The evaluation of `arbP2::(Bool,Bool)` (i.e., $arbP2\ \#bool\ \#bool$) obtains the same values as before. However, the evaluation of `arbL2::[Bool]` (i.e., $arbL2\ \#bool$) obtains the four expected values: $[true, true]$, $[true, false]$, $[false, true]$ and $[false, false]$. The following reduction shows the difference:

$$
\begin{array}{ll}
 & \underline{arbL2\ \#bool} \\
\to^{lf}_{(Fapp)} & [\underline{arb\ \#bool}, arb\ \#bool] \\
\to^{lf}_{(Fapp)} & [true, \underline{arb\ \#bool}] \\
\to^{lf}_{(Fapp)} & [true, false]
\end{array}
$$

5. PRACTICAL ISSUES

As we have seen, the proposed type system accepts more functions than usual Damas-Milner type system or the system in [20]. And this relaxation allows us to write type-indexed functions, programs with GADTs, generic functions or programs with existential types. However, the programmer may want to write functions with the usual Damas-Milner type and let the compiler to infer it. We propose to combine the type system presented here and that of [20] —that is equivalent to Damas-Milner for Haskell 98 like programs—, allowing the programmer to choose between them by means of annotations. Functions annotated with

@relaxed will be treated by the type system proposed here, therefore their type must be provided and it will be checked. Otherwise, they will be treated with the usual type system and their type will be inferred if not provided. This is similar to what it is done with GADTs, which are used to allow some program rules to have a more specific type than the type for the function they define. Instead of using GADT arguments to propagate this liberal typing to the arguments with which they share some type variable—as we did in the *eq* example from Sect. 4.1—, we specify this behaviour by means of these annotations, which affect to each function argument. Therefore it is easier to define functions for which we want a liberal behaviour for several independent arguments.

The combination is based in $\mathcal{B}$ [20], a procedure that given a set of assumptions and a program, calculates a type substitution π that makes the program well-typed (wrt. the notion in that paper). Theorem 3.1 assures that well-typed programs using the definition in [20] are also well-typed with the definition used in this paper, so we can use $\mathcal{B}$ to infer usual types for functions and these types will make the program well-typed. The global procedure works stratifically in each strongly connected component of the dependency graph in a topological order. If the functions in the component do not have annotations, $\mathcal{B}$ is used to infer the types of the whole block, and then they are generalized. Otherwise, the rules are checked using Definition 3.1. Notice that proceeding this way, the initial assumptions when type-checking a block of *relaxed* functions will be closed. Therefore the last point of the definition will hold trivially. For simplicity we assume that all the functions in a strongly connected component will have the annotation **@relaxed** or not, i.e., combinations of relaxed and usual functions are not allowed in the components.

The following example shows the method in a very simple program:

EXAMPLE 5.1. *Program with annotations*

```
@relaxed
eq :: A -> A -> bool
eq false false = true
eq false true  = true
eq z     z     = true
eq z    (s X)  = false

isFalse X = (eq X false)
```

isFalse *depends on* eq *but not viceversa. Therefore each one forms a strongly connected component.* eq *will be first treated. Since it has a* **@relaxed** *annotation its type will be checked using Definition 3.1. The rules of* eq *are well-typed wrt.* $\forall\alpha.\alpha \to \alpha \to bool$*, so* isFalse *will be treated next. As it does not have any annotation, the usual method will be used, inferring the type* $bool \to bool$*.*

Apart from functions, in this paper data constructor are also relaxed since they can have any type provided it is coherent with CS. For them, we propose a syntax similar to GADTs where the type of each one is given explicitly by the programmer.

6. CONCLUSIONS

Starting from a simple type system, essentially Damas-Milners's one, we have proposed a new notion of well-typed program that exhibits interesting properties. Let us give a short final summary and discussion of pros and cons of our proposal, some of them discussed more thoroughly before.

In the positive side: it is simple (needs only a check for each program rule); gives support to existential types and GADTs; opens new possibilities to genericity (f.i., the natural rules for equality and 'apply' are well-typed); has good properties (type safety); allows implementing type classes with good performance and respecting call-time-choice semantics; avoids unsafe uses of HO-patterns.

One negative aspect is that types must be declared for functions (but this is not so unusual; moreover see Section 5). Another point yet not discussed is the following: relaxing the well-typedness condition has the risk of accepting some expressions than one might prefer to detect and reject at compile time. This problem is always present in typed languages. For instance, in Haskell 98 the expression *head []* is well-typed, but its evaluation fails and produces a run-time error. Being more liberal when typing a program, we have more opportunities for such situations. Consider, for instance, our initial example of the function *size* with declared type $\forall\alpha.\alpha \to nat$. Any expression *size e*, for any well-typed *e*, is itself well-typed, even if *e*'s type is not considered in the definition of *size*, for instance, a functional type. We remark, however, that the case is a bit odder but, at least in our setting, not that different from *head []*: in both cases, the evaluation will fail since there is no applicable rule (and notice that in our system there *could* be rules for *size e*, even for functional *e*'s, thanks to HO-patterns). Moreover, failed evaluation in FLP should not be seen as errors, rather as useless branches in a possibly non-deterministic computation space. Would type classes or GADTs do any better? 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; as a consequence, 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 family of distinct representation types, which is a programming overhead somehow against genericity!

Anyway, this discussion lead us to the conclusion that our approach should not be regarded as opposed to type classes or GADTs, but rather as a complementary means. We find suggestive to think of the following scenario for our system TOY in the next future: type classes are added to the system and implemented through our generic functions; ordinary Damas-Milner typing and generic typing can be freely mixed using annotations as described in Section 5; 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.

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

• A formal specification of the translation of type classes into our generic functions following the ideas sketched in Section

4.4. It should cover also constructor classes and multiparameter type classes.

• Despite of the lack of principal types, some work on type inference can be done, in the spirit of [32].

• Combining our genericity with the existence of modules could require adopting *open* types and functions [19].

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

7. REFERENCES

[1] S. Antoy and A. P. Tolmach. Typed higher-order narrowing without higher-order strategies. *In Proc. FLOPS'99*, 335–353. Springer LNCS 1722, 1999.

[2] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A Call-by-need Lambda Calculus. In *Proc. POPL'95*, 233–246. ACM, 1995.

[3] L. Augustsson. Implementing haskell overloading. In *Proc. FPCA'93*, 65–73. ACM, 1993.

[4] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, July 2003.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. POPL'82*, 207–212. ACM, 1982.

[6] A. Gerdes. Comparing generic programming libraries. Master's thesis, Dep. Computer Science, Open University, 2007.

[7] J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. ICLP'97*, 153–167. MIT Press, 1997.

[8] J. C. Gonzalez-Moreno, M. T. Hortala-Gonzalez, and M. Rodriguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1), July 2001.

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

[10] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in haskell. *ACM TOPLAS*, 18(2):109–138, 1996.

[11] M. Hanus. Multi-paradigm declarative languages. In *Proc. ICLP'07*, 45–75. Springer LNCS 4670, 2007.

[12] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2), March 2006. Available at *http://www.informatik.uni-kiel.de/~curry/report.html*.

[13] M. Hanus (ed.). *PAKCS 1.8.1, The Portland Aachen Kiel Curry System, User manual*, 2007. `http://www.informatik.uni-kiel.de/~pakcs/Manual.pdf`.

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

[15] R. Hinze and A. Löh. Generic programming, now! In *Lecture notes for the Spring School on Datatype-Generic Programming 2006*, 150–208. Springer LNCS 4719, 2007.

[16] S. P. Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.

[17] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM TOPLAS*, 15(2):290–311, 1993.

[18] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM TOPLAS*, 16(5):1411–1430, 1994.

[19] A. Löh and R. Hinze. Open data types and open functions. In *Proc. PPDP'06*, 133–144. ACM, 2006.

[20] F. J. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. Advances in type systems for functional logic programming. In *Pre-proc. WFLP'09*, 157–171, June 2009.

[21] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. FLOPS'08*, 147–162. Springer LNCS 4989, 2008.

[22] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. RTA'99*, 244–247. Springer LNCS 1631, 1999.

[23] W. Lux. Adding haskell-style overloading to curry. In *Workshop of Working Group 2.1.4 of the German Computing Science Association GI*, 2008.

[24] W. Lux. 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`, August 2009.

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

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

[27] J. J. Moreno-Navarro, J. Mariño, A. del Pozo-Pietro, Á. Herranz-Nieva, and J. García-Martín. Adding type classes to functional-logic languages. In *Proc. APPIA-GULP-PRODE'96*, 427–438, 1996.

[28] M. Odersky and K. Läufer. Putting type annotations to work. In *Proc. POPL'96*, 54–67. ACM, 1996

[29] J. Peterson and M. Jones. Implementing type classes. In *Proc. PLDI'93*, 227–236. ACM, 1993.

[30] S. Peyton Jones. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge Univ. Press, 2003.

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

[32] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proc. ICFP'09*, 341–352. ACM, 2009.

[33] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL'89*, 60–76. ACM, 1989.

APPENDIX

A. MEASUREMENT OF THE SPEEDUP

For each system and subclass depth we have performed 100 speedup measurements. The mean of these speedups is the value which appears in Figure 8. For interpreted systems we have activated the corresponding flag to print the elapsed time after each expression evaluation. Hugs does not print any time information but the number of reductions, so we have used that information to calculate the speedup. Although it is not a real speedup, we consider that it shows faithfully the gain in efficiency obtained. For GHC, we have used the system command `time` to measure the time consumption of the compiled binary executable. These executables have been generated using the `-O2` option to enable optimizations.

Detailed results and source programs can be found in `http://gpd.sip.ucm.es/enrique/publications/genericFLP/speedup.zip`.

B. PROOFS AND AUXILIARY RESULTS

We first present some notions used in the proofs.

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

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

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

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

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

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

B.1 Auxiliary results

Theorem B.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 B.1 (SOUNDNESS OF $\Vdash$). $\mathcal{A} \Vdash e : \tau|\pi \implies \mathcal{A}\pi \vdash e : \tau$

Theorem B.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 the most general ones.

THEOREM B.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 B.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* $\mathcal{A} \vdash e : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$.

c) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash e : \tau$ *and* $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$ *then* $\mathcal{A} \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$.

PROOF. The proof of Theorems B.1, B.2 and B.3 appears in [25]. □

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

B.2 Proof of Theorem 3.1

PROOF. In [20] and this paper the definition of well-typed program proceeds rule by rule, so we only have to prove that if $wt^{old}_{\mathcal{A}}(f\ t_1 \ldots t_n \to e)$ then $wt_{\mathcal{A}}(f\ t_1 \ldots t_n \to e)$. For the sake of conciseness we will consider functions with just one argument: $f\ t \to e$. Since patterns are linear (all the variables are different) the proof for functions with more arguments follows the same ideas.

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

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

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

a) $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t : \tau_L|\pi_L$

b) $\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_R|\pi_R$

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

d) $\mathcal{A}\pi_L = \mathcal{A}$, $\mathcal{A}\pi_R = \mathcal{A}$, $\mathcal{A}\pi = \mathcal{A}$

By the type derivation of t and Theorem B.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 $f\ t$:

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

By Remark B.1 we are sure that $\mathcal{A}\pi_t = \mathcal{A}$. Since $\tau'_t \to \tau'_e$ is a variant of $\mathcal{A}(f)$ we know that it contains only free type variables in $\mathcal{A}$ or fresh variables, so $(\tau'_t \to \tau'_e)\pi_t = \tau'_t \to \tau'_e$. In order to complete the type inference we need to create a unifier π_u for $(\tau'_t \to \tau'_e)\pi_t$ and $\tau_t \to \gamma$, being γ a fresh type variable. Notice that by Theorem B.2 we know that $\mathcal{A}\pi_t\pi''_t = \mathcal{A}$ and by Remark B.1 $\mathcal{A}\pi_t = \mathcal{A}$, so $\mathcal{A}\pi''_t = \mathcal{A}$. Since $\tau'_t \to \tau'_e$ contains only type variables which are free in $\mathcal{A}$ or fresh type variables generated during the inference, π''_t will not affect it. Defining π_u as $\pi''_t|_{ftv(\tau_t)} + [\gamma/\tau'_e]$ we have

an unifier, since:

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

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 B.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 B.1 we also know that $\mathcal{A}\pi_e = \mathcal{A}$, so $\mathcal{A}\pi_e'' = \mathcal{A}$.

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

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

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

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

Finally we have to prove that **d)** $\mathcal{A}\pi_t\pi_g = \mathcal{A}$, $\mathcal{A}\pi_e = \mathcal{A}$ and $\mathcal{A}\pi = \mathcal{A}$. For the first case we already know that $\mathcal{A}\pi_t = \mathcal{A}$ and $\mathcal{A}\pi_t'' = \mathcal{A}$. Since π_g is defined as $\pi_t''|_{ftv(\tau_t)} + [\gamma/\tau_e']$ and γ is a fresh type variable not appearing in $ftv(\mathcal{A})$ then $\mathcal{A}\pi_t\pi_g = \mathcal{A}\pi_g = \mathcal{A}\pi_t''|_{ftv(\tau_t)} = \mathcal{A}$. For the second case, $\mathcal{A}\pi_e = \mathcal{A}$ holds using Remark B.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 $\mathcal{A}\pi = \mathcal{A}\pi_e'' = \mathcal{A}$. □

B.3 Proof of Subject Reduction: Theorem 3.2

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

(Fapp) If we reduce an expression e using the (Fapp) rule is because e has the form $f\ t_1\theta \dots t_n\theta$ (being $f\ t_1 \dots t_m \to r$ 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 $wt_{\mathcal{A}}(f\ t_1 \dots t_m \to e)$, and by the definition of well-typed rule (Definition 3.1) we have:

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

(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

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

where $\theta = [\overline{X_n/t_n'}]$. Since the type derivation *(E)* exists, then there exists also a type derivation for each pattern t_i': *(F)* $\mathcal{A} \vdash t_i' : \tau_i$.

If we replace every pattern t_i' in the type derivation *(E)* by their associated variable X_i and we add the assumptions $\{\overline{X_n : \tau_n}\}$ to $\mathcal{A}$, we obtain the type derivation:

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

By *(A)* and *(G)* and Theorem B.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 B.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 B.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 B.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 B.3-b) we can add the type assumptions $\{\overline{X_n : \tau_n}\}$ to the type derivations in *(F)*, obtaining *(Q)* $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t_i' : \tau_i$.

By Theorem B.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)*:

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

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

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

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

The rest of the cases are the same as the proof in [25] □

B.4 Theorem 3.3: Progress

For the proof of Progress we will need a result stating that *junk* expressions cannot have a valid type wrt. any coherent set of assumptions $\mathcal{A}$.

LEMMA B.1. *If e is a* junk *expression* wrt. *a set of constructor symbols CS then there are not any $\mathcal{A}$ and type τ such that $\mathcal{A}$ is coherent with CS and $\mathcal{A} \vdash e : \tau$.*

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

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

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

Using the previous result we can now prove the Progress.

Proof of Progress: Theorem 3.3

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 $(f \to r) \in \mathcal{P}$ then we can apply rule (Fapp), so $\mathcal{P} \vdash s \to^{lf} r$. Otherwise there is not any rule $(l \to r) \in \mathcal{P}$ such that l and f unify, so we can apply the rule for the matching failure (Ffail) obtaining $\mathcal{P} \vdash s \to^{lf} fail$.

Inductive Step

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

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

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

- X) This cannot happen because e_1 is ground.
- $c\ t_1 \dots t_n$ 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 $c\ t_1 \dots t_n\ e_2$ with $n+1 \leq m$, which is a pattern.
  - $n = m$)
    - If $c = fail$ then $m = n = 0$, so we have the expression $fail\ e_2$. In this case we can apply rule (FailP), so $\mathcal{P} \vdash fail\ e_2 \to^{lf} fail$.
    - Otherwise $e_1\ e_2$ is $c\ t_1 \dots t_n\ e_2$ with $n+1 > m$, which is *junk*. This cannot happen because $\mathcal{A}$ is coherent and $\mathcal{A} \vdash e_1\ e_2 : \tau$, and Lemma B.1 states that *junk* expressions cannot be well-typed wrt. a coherent set of assumptions.
- $f\ t_1 \dots t_n$ 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 $f\ t_1 \dots t_n\ e_2$ which is a partially applied function symbol, i.e., a pattern.
  - $n + 1 = m$) Then $e_1\ e_2$ is $f\ t_1 \dots t_n\ e_2$. If there is a rule $(l \to r) \in \mathcal{P}$ such that $l\theta = f\ t_1 \dots t_n\ e_2$ then we can apply rule (Fapp), so $\mathcal{P} \vdash e_1\ e_2 \to^{lf} r\theta$. If such a rule does not exist, then there is not any rule $(l' \to r') \in \mathcal{P}$ such that l' and $f\ t_1 \dots t_n\ e_2$ unify. Notice that since $f\ t_1 \dots t_n\ e_2$ is ground it does not have variables, so in this case pattern matching and unification are equivalent. Therefore we can apply the rule for the matching failure (Ffail) obtaining $\mathcal{P} \vdash e_1\ e_2 \to^{lf} fail$.

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

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

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

- e_1 is a pattern) Then we can use the (Bind) rule, obtaining $\mathcal{P} \vdash let\ X = e_1\ in\ e_2 \to^{lf} e_2[X/e_1]$.
- e_1 is not a pattern) Since $let\ X = e_1\ in\ e_2$ is ground we know that e_1 is ground (notice that this does not force e_2 to be ground). Moreover, $\mathcal{A} \vdash e_1 : \tau_t$, so by the Induction Hypothesis we can rewrite e_1 to some e'_1: $\mathcal{P} \vdash e_1 \to^{lf} e'_1$. Using the (Contx) rule we can transform this local step into a step in the whole expression: $\mathcal{P} \vdash let\ X = e_1\ in\ e_2 \to^{lf} let\ X = e'_1\ in\ e_2$.

□