Advances in Type Systems for Functional Logic Programming *

Francisco J. López-Fraguas
Enrique Martín-Martín
Juan Rodríguez-Hortalá

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

Abstract. Type systems are widely used in programming languages as a powerful tool providing safety to programs, and forcing the programmers to write code in a clearer way. Functional logic languages have inherited Damas & Milner type system from their functional part due to its simplicity and popularity. In this paper we address a couple of aspects that can be subject of improvement. One is related to a problematic feature of functional logic languages not taken under consideration by standard systems: it is known that the use of opaque HO patterns in left-hand sides of program rules may produce undesirable effects from the point of view of types. We re-examine the problem, and propose a Damas & Milner-like type system where certain uses of HO patterns (even opaque) are permitted while preserving type safety, as proved by a subject reduction result that uses HO-let-rewriting, a recently proposed reduction mechanism for HO functional logic programs. The other aspect is the different ways in which polymorphism of local definitions can be handled. At the same time that we formalize the type system, we have made the effort of technically clarifying the overall process of type inference in a whole program.

1 Introduction

Type systems for programming languages are an active area of research [18], no matter which paradigm one considers. In the case of functional programming, most type systems have arisen as extensions of Damas & Milner’s [3], for its remarkable simplicity and good properties (decidability, existence of principal types, possibility of type inference). Functional logic languages [11, 7, 6], in their practical side, have inherited more or less directly Damas & Milner’s types. In principle, most of the type extensions proposed for functional programming could be also incorporated to functional logic languages (this has been done, for instance, for type classes in [15]). However, if types are not only decoration but

* This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03), STAMP (TIN2008-06622-C03-01), Promesas-CAM (S-0505/TIC/0407) and GPD-UCM (UCM-BSCH-GR58/08-910502)
are to provide safety, one should be sure that the adopted system has indeed good properties. In this paper we tackle a couple of orthogonal aspects of existing FLP systems that are problematic or not well covered by standard Damas & Milner systems. One is the presence of so called HO patterns in programs, an expressive feature allowed in some systems and for which a sensible semantics exists [4]; however, it is known that unrestricted use of HO patterns leads to type unsafety, as recalled below. The second is the degree of polymorphism assumed for local pattern bindings, a matter with respect to which existing FP or FLP systems vary greatly.

The rest of the paper is organized as follows. The next two subsections further discuss the two mentioned aspects. Sect. 2 contains some preliminaries about FL programs and types. In Sect. 3 we expose the type system and prove its soundness wrt. the let rewriting semantics of [10]. Sect. 4 contains a type inference relation, which let us find the most general type of expressions. Sect. 5 presents a method to infer types for programs. Finally, Sect. 6 contains some conclusions and future work. Omitted proofs can be found in [12].

1.1 Higher order patterns

In our formalism patterns appear in the left-hand side of rules and in lambda or let expressions. Some of these patterns can be HO patterns, if they contain partial applications of function or constructor symbols. HO patterns can be a source of problems from the point of view of the types. In particular, it was shown in [5] that unrestricted use of HO patterns leads to loss of subject reduction, an essential property for a type system expressing that evaluation does not change types. The following is a crisp example of the problem.

Example 1 (Polymorphic Casting [2]). Consider the program consisting of the rules \( \text{snd} \ X \ Y \rightarrow Y \), and \( \text{true} \ X \rightarrow X \), and \( \text{false} \ X \rightarrow \text{false} \), with the usual types inferred by a classical Damas & Milner algorithm. Then we can write the functions \( \text{co} \ (\text{snd} \ X) \rightarrow X \) and \( \text{cast} \ X \rightarrow \text{co} \ (\text{snd} \ X) \), whose inferred types will be \( \forall \alpha. \forall \beta. (\alpha \rightarrow \alpha) \rightarrow \beta \) and \( \forall \alpha. \forall \beta. \alpha \rightarrow \beta \) respectively. It is clear that the expression \( \text{and} \ (\text{cast} \ 0) \ \text{true} \) is well-typed, because \( \text{cast} \ 0 \) has type \( \text{bool} \) (in fact it has any type), but if we reduce that expression using the rule of \( \text{cast} \) the resulting expression \( \text{and} \ 0 \ \text{true} \) is ill-typed.

The problem arises when dealing with HO patterns, because unlike FO patterns, knowing the type of a HO pattern does not always permit us to know the type of its subpatterns. In the previous example the cause is function \( \text{co} \), because its pattern \( \text{snd} \ X \) is opaque and shadows the type of its subpattern \( X \). Usual inference algorithms treat this opacity as polymorphism, and that is the reason why it is inferred a completely polymorphic type for the result of the function \( \text{co} \).

In [5] the appearance of any opaque pattern in the left-hand side of the rules is prohibited, but we will see that it is possible to be less restrictive. The key is making a distinction between transparent and opaque variables of a pattern:
a variable is transparent if its type is univocally fixed by the type of the pattern, and is opaque otherwise. We call a variable of a pattern critical if it is opaque in the pattern and also appears elsewhere in the expression. The formal definition of opaque and critical variables will be given in Sect. 3. With these notions we can relax the situation in [5], prohibiting only those patterns having critical variables.

1.2 Local definitions

Functional and functional logic languages provide syntax to introduce local definitions inside an expression. But in spite of the popularity of let-expressions, different implementations treat them differently because of the polymorphism they give to bound variables. This difference can be observed in Ex. 2, being \((e_1, \ldots, e_n)\) and \([e_1, \ldots, e_n]\) the usual tuple and list notation respectively.

Example 2 (let expressions). Let \(e_1\) be let \(F = \text{id}\) in \((F \text{true}, F \text{0})\), and \(e_2\) be let \([F, G] = [\text{id}, \text{id}]\) in \((F \text{true}, F \text{0}, G \text{0}, G \text{false})\)

Intuitively, \(e_1\) gives a new name to the identity function and uses it twice with arguments of different types. Surprisingly, not all implementations consider this expression as well-typed, and the reason is that \(F\) is used with different types in each appearance: \(\text{bool} \to \text{bool}\) and \(\text{int} \to \text{int}\). Some implementations as Clean 2.2, PAKCS 1.9.1 or KICS 0.81893 consider that a variable bound by a let-expression must be used with the same type in all the appearances in the body of the expression. In this situation we say that lets are completely monomorphic, and write \(\text{let}\_m\) for it.

On the other hand, we can consider that all the variables bound by the let-expression may have different but coherent types, i.e., are treated polymorphically. Then expressions like \(e_1\) or \(e_2\) would be well-typed. This is the decision adopted by Hugs Sept. 2006, OCaml 3.10.2 or F# Sept. 2008. In this case, we will say that lets are completely polymorphic, and write \(\text{let}\_p\) for it.

Finally, we can treat the bound variables monomorphically or polymorphically depending on the form of the pattern. If the pattern is a variable, the let treats it polymorphically, but if it is compound the let treats all the variables monomorphically. This is the case of GHC 6.8.2, SML of New Jersey v110.67 or Curry Münster 0.9.11. In this implementations \(e_1\) is well-typed, while \(e_2\) not. We call this kind of let-expression \(\text{let}\_pm\).

Fig. 1 summarizes the decisions of various implementations of functional and functional logic languages. The exact behavior wrt. types of local definitions is usually not well documented, not to say formalized, in those systems. One of our contributions is this paper is to technically clarify this question by adopting a neutral position, and formalizing the different possibilities for the polymorphism of local definitions.
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 \( n \). We also assume a denumerable set \( DV \) of data variables \( X \). We define the set of patterns \( Pat \ni t ::= X \mid c t_1 \ldots t_n \ (n \leq k) \mid f t_1 \ldots t_n \ (n < k) \), where \( c \in DC^k \) and \( f \in FS^k \); and the set of expressions \( Exp \ni e ::= X \mid c \mid f \mid e_1 e_2 \mid \lambda t.e \mid let_m t = e_1 \ in \ e_2 \mid let_p t = e_1 \ in \ e_2 \mid let_p t = e_1 \ in \ e_2 \) where \( c \in DC \) and \( f \in FS \). We split the set of patterns in two: first order patterns \( FOPat \ni fot ::= X \mid c t_1 \ldots c_n \) where \( c \in DC^n \), and higher order patterns \( HOPat = Pat \setminus FOPat \). Expressions \( h e_1 \ldots e_n \) are called junk if \( h \in CS^k \) and \( n > k \), and active if \( h \in FS^k \) and \( n \geq k \). \( FV(e) \) is the set of variables in \( e \) which are not bound by any lambda or let expression and is defined in the usual way (notice that since our let expressions do not support recursive definitions the bindings of the pattern only affect \( e_2 \): \( FV(let_s t = e_1 \ in \ e_2) = FV(e_2) \cup (FV(e_2) \setminus var(t)) \). A one-hole context \( C \) is an expression with exactly one hole. A data substitution \( \theta \in PSubst \) is a finite mapping from data variables to patterns: \( [X_i/t_i] \). Substitution application over data variables and expressions is defined in the usual way. A program rule is defined as \( PRule \ni \rho ::= f t_1 \ldots t_n \rightarrow e \ (n \geq 0) \) where the set of patterns \( \tau \) is linear and \( FV(e) \subseteq \bigcup_i \{\text{var}(t_i)\} \). Therefore, extra variables are not considered in this paper. A program is a set of program rules \( \text{Prog} \ni \mathcal{P} ::= \{\rho_1; \ldots; \rho_n\} \ (n \geq 0) \).

For the types we assume a denumerable set \( TV \) of type variables \( \alpha \) and a countable alphabet \( TC = \bigcup_{n \in \mathbb{N}} TC^n \) of type constructors \( C \). The set of simple types is defined as \( SType \ni \tau ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid C \tau_1 \ldots \tau_n \ (C \in TC^n) \). Based on simple types we define the set of type-schemes as \( TScheme \ni \sigma ::= \tau \mid \forall \alpha.\sigma \). The set of free type variables (FTV) of a simple type \( \tau \) is \( var(\tau) \), and for type-schemes \( FTV(\forall \alpha.\tau) = FTV(\tau) \setminus \{\alpha\} \). A type-scheme \( \forall \alpha_1\ldots\alpha_n \rightarrow \tau \) is transparent if \( FTV(\alpha_i) \subseteq FTV(\tau) \). A set of assumptions \( \mathcal{A} \) is \( \{s_i : \alpha_i\} \), where \( s_i \in DC \cup FS \cup DV \). Notice that the transparency of type-schemes for data constructors

<table>
<thead>
<tr>
<th>Programming language and version</th>
<th>let_m</th>
<th>let_pm</th>
<th>let_p</th>
</tr>
</thead>
<tbody>
<tr>
<td>GHC 6.8.2</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hugs Sept. 2006</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Standard ML of New Jersey 110.67</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocam 3.10.2</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>F# Sept. 2008</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clean 2.0</td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>TOY 2.3.1*</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Curry PAKCS 1.9.1</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Curry Münster 0.9.11</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KICS 0.81893</td>
<td>×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Let expressions in different programming languages.
is not required in our setting, although that hypothesis is usually assumed in classical Damas & Milner type systems. If \((s_i : \sigma_i) \in \mathcal{A}\) we write \(\mathcal{A}(s_i) = \sigma_i\). A type substitution \(\pi \in TS\text{ubst}\) is a finite mapping from type variables to simple types \([\alpha_i/\tau_i]\). For sets of assumptions \(FTV(\{\[\alpha_i/\tau_i]\}) = \bigcup_i FTV(\sigma_i)\). We will say a type-scheme \(\sigma\) is closed if \(FTV(\sigma) = \emptyset\). 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 \(\sigma\) is an instance of \(\sigma'\) if \(\sigma = \sigma'\pi\) for some \(\pi\). \(\tau'\) is a generic instance of \(\sigma \equiv \forall \tau.\tau\) if \(\tau' = \tau[\alpha_i/\tau_i]\) for some \(\tau_i\), and we write it \(\sigma \triangleright \tau'\). We extend \(\triangleright\) to a relation between type-schemes by saying that \(\sigma \triangleright \sigma'\) iff every simple type such that is a generic instance of \(\sigma'\) is also a generic instance of \(\sigma\). Then \(\forall \tau_i.\tau \triangleright \forall \tau_i.\tau[\alpha_i/\tau_i]\) iff \(\{\beta_i\} \cap FTV(\forall \tau_i.\tau) = \emptyset\) [13]. Finally, \(\tau'\) is a variant of \(\sigma \equiv \forall \tau.\tau\) (\(\sigma \triangleright_{\text{var}} \tau'\)) if \(\tau' = \tau[\alpha_i/\beta_i]\) and \(\beta_i\) are fresh type variables.

### 3 Type derivation

We propose a modification of Damas & Milner type system [3] with some differences. We have found convenient to separate the task of giving a regular Damas & Milner type and the task of checking critical variables. To do that we have defined two different type relations: \(\vdash\) and \(\vdash^\bullet\).

The basic typing relation \(\vdash\) in the upper part of Fig. 2 is like the classical Damas & Milner’s system but extended to handle the three different kinds of let expressions and the occurrence of patterns instead of variables in lambda and let expressions. We have also made the rules more syntax-directed so that the form of type derivations depends only on the form of the expression to be typed. \(Gen(\tau, \mathcal{A})\) is the closure or generalization of \(\tau\) wrt. \(\mathcal{A}\) [3, 13, 19], which generalizes all the type variables of \(\tau\) that do not appear free in \(\mathcal{A}\). Formally: \(Gen(\tau, \mathcal{A}) = \forall \{\tau_i\} = FTV(\tau) \setminus FTV(\mathcal{A})\). As can be seen, \([\text{LET}_m]\) \ and \([\text{LET}^h_{pm}]\) behave the same, and do not generalize any of the types \(\tau_i\) for the variables \(X_i\) to give a type for the body. On the contrary, \([\text{LET}^X_{pm}]\) and \([\text{LET}_p]\) generalize the types given to the variables. Notice that if two variables share the same type in the set of assumptions \(\mathcal{A}\), generalization will lose the connection between them. This fact can be seen with \(e_2\) in Ex. 2. Although the type for both \(F\) and \(G\) can be \(\alpha \rightarrow \alpha\) (with \(\alpha\) a variable not appearing in \(\mathcal{A}\)) the generalization step will assign both the type-scheme \(\forall \alpha.\alpha \rightarrow \alpha\), losing the connection between them.

The \(\vdash^\bullet\) relation (lower part of Fig. 2) uses \(\vdash\) but enforces also the absence of critical variables. A variable \(X_i\) is opaque in \(t\) when it is possible to build a type derivation for \(t\) where the type assumed for \(X_i\) contains type variables which do not occur in the type derived for the pattern. The formal definition is as follows.

**Definition 1 (Opaque variable of \(t\) wrt. \(\mathcal{A}\)).** Let \(t\) be a pattern that admits type wrt. a given set of assumptions \(\mathcal{A}\). We say that \(X_i \in \mathcal{X}_i = \text{var}(t)\) is opaque wrt. \(\mathcal{A}\) iff \(\exists \tau, \tau\) s.t. \(\mathcal{A} \cup \{X_i : \tau_i\} \vdash^\bullet t : \tau\) and \(FTV(\tau_i) \notin FTV(\tau)\).
The previous definition is based on the existence of a certain type derivation, and therefore cannot be used as an effective check for the opacity of variables. Prop. 1 provides a more operational characterization of opacity that exploits the close relationship between $\vdash$ an type inference $\models$ presented in Sect. 4.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ID]</td>
<td>$A \vdash s : \tau$ if $s \in DC \cup FS \cup DV$ $\land (s : \sigma) \in A \land \sigma \gg \tau$</td>
</tr>
<tr>
<td>[APP]</td>
<td>$A \vdash e_1 : \tau_1$ $\vdash e_2 : \tau_1$ $\vdash e_1 e_2 : \tau$</td>
</tr>
<tr>
<td>[A]</td>
<td>$A \vdash e_1 : \tau_1$ $\vdash {X_i : \tau_i} \vdash t : \tau_1$ $\vdash \lambda x : \tau_2 \vdash \tau_2$ $\vdash \lambda x : \tau_2 \vdash \tau_2$</td>
</tr>
<tr>
<td>[LET_m]</td>
<td>$A \vdash e_1 : \tau_1$ $\vdash {X_i : \tau_i} \vdash e_2 : \tau_2$ $\vdash \lambda x : \tau_2 \vdash \tau_2$ $\vdash \lambda x : \tau_2 \vdash \tau_2$</td>
</tr>
<tr>
<td>[LET_m]</td>
<td>$A \vdash e_1 : \tau_1$ $\vdash {X_i : \tau_i} \vdash t : \tau_1$ $\vdash {X_i : \tau_i} \vdash \tau_1$ $\vdash {X_i : \tau_i} \vdash \tau_1$</td>
</tr>
<tr>
<td>[LET_m]</td>
<td>$A \vdash e_1 : \tau_1$ $\vdash {X_i : \tau_i} \vdash e_2 : \tau_2$ $\vdash \lambda x : \tau_2 \vdash \tau_2$ $\vdash \lambda x : \tau_2 \vdash \tau_2$</td>
</tr>
<tr>
<td>[LET_m]</td>
<td>$A \vdash e_1 : \tau_1$ $\vdash {X_i : \tau_i} \vdash {X_i : \tau_i} = {X_i : \tau_i}$ $\vdash {X_i : \tau_i} \vdash \tau_1$ $\vdash {X_i : \tau_i} \vdash \tau_1$</td>
</tr>
<tr>
<td>[LET_m]</td>
<td>$A \vdash e_1 : \tau_1$ $\vdash {X_i : \tau_i} \vdash {X_i : \tau_i} = {X_i : \tau_i}$ $\vdash {X_i : \tau_i} \vdash \tau_1$ $\vdash {X_i : \tau_i} \vdash \tau_1$</td>
</tr>
<tr>
<td>[LET_m]</td>
<td>$A \vdash e_1 : \tau_1$ $\vdash {X_i : \tau_i} \vdash {X_i : \tau_i} = {X_i : \tau_i}$ $\vdash {X_i : \tau_i} \vdash \tau_1$ $\vdash {X_i : \tau_i} \vdash \tau_1$</td>
</tr>
<tr>
<td>[P]</td>
<td>$A \vdash e : \tau$ $\vdash \cdot e : \tau$ if $\text{critVar}_A(e) = \emptyset$</td>
</tr>
</tbody>
</table>

**Fig. 2. Rules of type system**

**Proposition 1.** $X_i \in \overline{X_i} = \text{var}(t)$ is opaque wrt. $A$ if $A \vdash \{X_i : \alpha_i\}$ $\vdash t : \tau_g | \pi_g$ and $\text{FTV}(\alpha_i \pi_g) \not\subseteq \text{FTV}(\tau_g)$.

We write $\text{opaqueVar}_A(t)$ for set of opaque variables of $t$ wrt. $A$. Now, we can define the critical variables of an expression $e$ wrt. $A$ as those variables that, being opaque in a let or lambda pattern of $e$, are indeed used in $e$. Formally:
Definition 2 (Critical variables).
\[ \text{critVar}_A(s) = \emptyset \text{ if } s \in DC \cup FS \cup DV \]
\[ \text{critVar}_A(e_1e_2) = \text{critVar}_A(e_1) \cup \text{critVar}_A(e_2) \]
\[ \text{critVar}_A(\lambda t.e) = (\text{opaqueVar}_A(t) \cap \text{FV}(e)) \cup \text{critVar}_A(e) \]
\[ \text{critVar}_A(\text{let } t = e_1 \text{ in } e_2) = (\text{opaqueVar}_A(t) \cap \text{FV}(e_2)) \cup \text{critVar}_A(e_1) \cup \text{critVar}_A(e_2) \]

Notice that if we write the function \( co \) of Ex. 1 as \( \lambda (snd \ X).X \), it is well-typed \( \vdash \) using the usual type for \( \text{snd} \). However it is ill-typed \( \vdash^* \) since \( X \) is an opaque variable in \( snd \ X \) and it occurs in the body, so it is critical.

The typing relation \( \vdash^* \) has been defined in a modular way in the sense that the opacity check is kept separated from the regular Damas & Milner typing. Therefore it is easy to see that if every constructor and function symbol in program has a transparent assumption, then all the variables in patterns will be transparent, and so \( \vdash^* \) will be equivalent to \( \vdash \). This happens in particular for those programs using only first order patterns and whose constructor symbols come from a Haskell (or Toy, Curry)-like data declaration.

3.1 Properties of the typing relations

The typing relations fulfill a set of useful properties. Here we use \( \vdash^? \) for any of the two typing relations: \( \vdash \) or \( \vdash^* \).

Theorem 1 (Properties of the typing relations).
1. If \( \vdash^? e : \tau \) then \( \vdash^? e : \tau \pi, \) for any \( \pi \in TS\text{ubst} \).
2. Let \( s \in DC \cup FS \cup DV \) be a symbol not occurring in \( e \). Then \( \vdash^? e : \tau \iff \vdash^? e : \tau \).
3. If \( \vdash^? \{ X : \tau_x \} \vdash^? e : \tau \) and \( \vdash^? \{ X : \tau_x \} \vdash^? e' : \tau \) then \( \vdash^? \{ X : \tau_x \} \vdash^? e[X/e'] : \tau \).
4. If \( \vdash^? \{ s : \sigma \} \vdash e : \tau \) and \( \sigma' \succ \sigma \), then \( \vdash^? \{ s : \sigma' \} \vdash e : \tau \).

Part a) states that type derivations are closed under type substitutions. b) shows that type derivations for \( e \) depend only on the assumptions for the symbols in \( e \). c) is a substitution lemma stating that in a type derivation we can replace a variable by an expression with the same type. Finally, d) establishes that from a valid type derivation we can change the assumption of a symbol for a more general type-scheme, and we still have a correct type derivation for the same type. Notice that this is not true wrt. the typing relation \( \vdash^* \) because a more general type can introduce opacity. For example the variable \( X \) is opaque in \( snd \ X \) with the usual type for \( \text{snd} \), but with a more specific type such as \( \text{bool} \to \text{bool} \to \text{bool} \) it is no longer opaque.

3.2 Subject Reduction

Subject reduction is a key property for type systems, meaning that evaluation does not change the type of an expression. This ensures that run-time type errors will not occur. Subject reduction is only guaranteed for well-typed programs, a notion that we formally define now.
Definition 3 (Well-typed program). A program rule \( f \, t_1 \ldots t_n \rightarrow e \) is well-typed wrt. \( A \) if \( A \vdash \lambda t_1 \ldots \lambda t_n . e : \tau \) and \( \tau \) is a variant of \( A(f) \). A program \( P \) is well-typed wrt. \( A \) if all its rules are well-typed wrt. \( A \). If \( P \) is well-typed wrt. \( A \) we write \( wt_A(P) \).

Notice the use of the extended typing relation \( \vdash \) in the previous definition. This is essential, as we will explain later. Returning to Ex. 1, we can see that the program will not be well-typed because of the rule \( co\,(snd\,X) \rightarrow X \), since \( \lambda(snd\,X).X \) will be ill-typed wrt. the usual type for \( snd \), as we explained before.

Although the restriction that the type of the lambda abstraction associated to a rule must be a variant of the type of the function symbol (and not an instance) might seem strange, it is necessary. Otherwise, the fact that a program is well-typed will not give us important information about the functions like the type of their arguments, and will make us to consider as well-typed undesirable programs like \( P \equiv \{ f\,true \rightarrow true; f\,2 \rightarrow false \} \) with the assumptions \( A \equiv \{ f : \forall a.a \rightarrow bool \} \). Besides, this restriction is implicitly considered in [5].

\[
\begin{align*}
TRL(s) &= s, \text{ if } s \in DC \cup FS \cup DV \\
TRL(e_1, e_2) &= TRL(e_1) \cap TRL(e_2) \\
TRL(let_K X = e_1 in e_2) &= let_K X = TRL(e_1) \cap TRL(e_2), \text{ with } K \in \{m,p\} \\
TRL(let_{pm} X = e_1 in e_2) &= let_{pm} X = TRL(e_1) \cap TRL(e_2) \\
TRL(let_{m} t = e_1 in e_2) &= let_{m} Y = TRL(e_1) \cap TRL(e_2) \\
TRL(let_{p} t = e_1 in e_2) &= let_{p} Y = TRL(e_1) \cap TRL(e_2)
\end{align*}
\]

for \( \{X\} = \text{var}(t) \cap \text{var}(e_2), f_{X_i} \in FS \) fresh defined by the rule \( f_{X_i} : t \rightarrow X_i, Y \in DV \) fresh, \( t \) a non variable pattern.

Fig. 3. Transformation rules of let expressions with patterns

For subject reduction to be meaningful, a notion of evaluation is needed. In this paper we consider the let-rewriting relation of [10]. As can be seen, let-rewriting does not support let expressions with compound patterns. Instead of extending the semantics with this feature we propose a transformation from let-expressions with patterns to let-expressions with only variables (Fig. 3). There are various ways to perform this transformation, which differ in the strictness of the pattern matching. We have chosen the alternative explained in [17] that does not demand the matching if no variable of the pattern is needed, but otherwise forces the matching of the whole pattern. This transformation has been enriched with the different kinds of let expressions in order to preserve the types, as is stated in Th. 2. Notice that the result of the transformation and the expressions accepted by let-rewriting only has let\(_{m}\) or let\(_{p}\) expressions, since without compound patterns let\(_{pm}\) is the same as let\(_{p}\). Finally, we have added polymorphism annotations to let expressions (Fig. 4). Original \( \text{Flat} \) rule has been split into two, one for each kind of polymorphism. Although both behave the same from
the point of view of values, the splitting is needed to guarantee type preservation. λ-abstractions have been omitted, since they are not supported by let-rewriting.

(Fapp) \( f \, t_1 \ldots t_n \, \lambda \theta \rightarrow r \theta, \) if \((f \, t_1 \ldots t_n \rightarrow r) \in P \) and \( \theta \in P_{\text{Subst}} \)

(LetIn) \( e_1 \, e_2 \rightarrow^l \text{let}_m \, X = e_2 \, \text{in} \, e_1 \, X, \) if \( e_2 \) is an active expression, variable application, junk or let rooted expression, for \( X \) fresh.

(Bind) \( \text{let}_K \, X = t \, \text{in} \, e \rightarrow^l e[X/t], \) if \( t \in \text{Pat} \)

(Elim) \( \text{let}_K \, X = e_1 \, \text{in} \, e_2 \rightarrow^l e_2, \) if \( X \not\in \text{FV}(e_2) \)

(Flat\(_m\)) \( \text{let}_m \, X = (\text{let}_K \, Y = e_1 \, \text{in} \, e_2) \, \text{in} \, e_3 \rightarrow^l \text{let}_K \, Y = e_1 \, \text{in} \, (\text{let}_m \, X = e_2 \, \text{in} \, e_3), \) if \( Y \not\in \text{FV}(e_3) \)

(Flat\(_p\)) \( \text{let}_p \, X = (\text{let}_K \, Y = e_1 \, \text{in} \, e_2) \, \text{in} \, e_3 \rightarrow^l \text{let}_p \, Y = e_1 \, \text{in} \, (\text{let}_p \, X = e_2 \, \text{in} \, e_3) \)

(LetAp) \( (\text{let}_K \, X = e_1 \, \text{in} \, e_2) \, e_3 \rightarrow^l \text{let}_K \, X = e_1 \, \text{in} \, e_2 \, e_3, \) if \( X \not\in \text{FV}(e_3) \)

(Contx) \( C[e] \rightarrow^l C[e'], \) if \( C \not\in \emptyset, \) \( e \rightarrow^l e' \) using any of the previous rules

where \( K \in \{m, p\} \)

Fig. 4. Higher order let-rewriting relation \( \rightarrow^l \)

**Theorem 2 (Type preservation of the let transformation).** Assume \( A \vdash^* e : \tau \) and let \( P \equiv \{ f_{X_i} \, t_i \rightarrow X_i \} \) be the rules of the projection functions needed in the transformation of \( e \) according to Fig. 3. Let also \( A' \) be the set of assumptions over that functions, defined as \( A' \equiv \{ f_{X_i} : \text{Gen}(\pi_{X_i}, A) \} \), where \( A \equiv^* \lambda t_i, X_i : \tau_{X_i} \mid \pi_{X_i} \). Then \( A \oplus A' \vdash^* \text{TRL}(e) : \tau \) and \( wt_{A \oplus A'}(P) \).

Th. 2 also states that the projection functions are well-typed. Then if we start from a well-typed program \( P \) wrt. \( A \) and apply the transformation to all its rules, the program extended with the projections rules will be well-typed wrt. the extended assumptions: \( wt_{A \oplus A'}(P \cup P') \). This result is straightforward, because \( A' \) does not contain any assumption for the symbols in \( P \), so \( wt_{A}(P) \) implies \( wt_{A \oplus A'}(P) \).

Th. 3 states the subject reduction property for a let-rewriting step, but its extension to any number of steps is trivial.

**Theorem 3 (Subject Reduction).** If \( A \vdash^* e : \tau \) and \( wt_{A}(P) \) and \( P \vdash e \rightarrow^l e' \) then \( A \vdash^* e' : \tau \).

For this result to hold it is essential that the definition of well-typed program relies on \( \vdash^* \). A counterexample can be found in Ex. 1, where the program would be well-typed wrt. \( \vdash \) but the subject reduction property fails for \( \text{and} (\text{cast} \, 0) \, \text{true} \).

The proof of the subject reduction property is based on the following lemma, an important auxiliary result about the instantiation of transparent variables.
Intuitively it states that if we have a pattern $t$ with type $\tau$ and we change its variables by other expressions, the only way to obtain the same type $\tau$ for the substituted pattern is by changing the transparent variables for expressions with the same type. This is not guaranteed with opaque variables, and that is why we forbid their use in expressions.

**Lemma 1.** Assume $A \oplus \{X_i : \tau_i\} \vdash t : \tau$, where $\text{var}(t) \subseteq \{X_i\}$. If $A \vdash t[X_i/s_i] : \tau$ and $X_j$ is a transparent variable of $t$ wrt. $A$ then $A \vdash s_j : \tau_j$.

### 4 Type inference for expressions

The typing relation $\vdash$ lacks some properties that prevent its usage as a type-checker mechanism in a compiler for a functional logic language. First, in spite of the syntax-directed style, the rules for $\vdash$ and $\vdash \cdot$ have a bad operational behavior: at some steps they need to guess a type. Second, the types related to an expression can be infinite due to polymorphism. Finally, the typing relation needs all the assumptions for the symbols in order to work. To overcome these problems, type systems usually are accompanied with a type inference algorithm which returns a valid type for an expression and also establish the types for some symbols in the expression.

In this work we have given the type inference in Fig. 5 a relational style to show the similarities with the typing relation. But in essence, the inference rules represent an algorithm (similar to algorithm $W$ [3, 13]) which fails if any of the rules cannot be applied. This algorithm accepts a set of assumptions $A$ and an expression $e$, and returns a simple type $\tau$ and a type substitution $\pi$. Intuitively, $\tau$ will be the “most general” type which can be given to $e$, and $\pi$ the “minimum” substitution we have to apply to $A$ in order to be able to derive a type for $e$.

Th. 4 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 4 (Soundness of $\vdash \cdot$).** $A \vdash \cdot e : \tau|\pi \Rightarrow A[\pi] \vdash e : \tau$

Th. 5 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 5 (Completeness of $\vdash \cdot$ wrt $\vdash$).** If $A_\pi \vdash e : \tau'$ then $\exists \tau, \pi, \pi''. A \vdash e : \tau|\pi \land A[\pi]\pi'' = A[\pi'] \land \tau\pi'' = \tau'$.

A result similar to Th. 5 cannot be obtained for $\vdash \cdot$ because of critical variables, as the following example 3 shows.

**Example 3 (Inexistence of a most general typing substitution).** Let $A \equiv \{\text{snd} : \alpha \rightarrow \text{bool} \rightarrow \text{bool}\}$ and consider the following two valid derivations $D_1 \equiv A[\alpha/\text{bool}] \vdash \cdot \lambda(\text{snd} X).X : (\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool}$ and $D_2 \equiv A[\alpha/\text{int}] \vdash \cdot$
[ID] \[ A \models s : \tau \text{id} \] if \( s \in DC \cup FS \cup DV \) \& \((s : \sigma) \in A \wedge \sigma \succ \text{var} \tau \)

[APP] \[ A \models e_1 : \tau_1 \mid A \] \[ A \models e_2 : \tau_2 \mid A \]
\[ A \models e_1 e_2 : \alpha \pi \mid A \]
\[ \text{if} \ \alpha \text{ fresh type variable} \]
\[ \wedge \pi = \text{mgu}(\tau_1 \pi_2, \tau_2 \rightarrow \alpha) \]

[I\text{LET}] \[ A \rightarrow \{ X_i : \alpha_i \} \|\| e : \tau \mid A \]
\[ (A \oplus (X_i : \alpha_i)) \mid\| e : \tau_1 \mid A \]
\[ \text{if} \ \{ X_i \} = \text{var}(t) \]
\[ \wedge \pi = \text{mgu}(\tau_1 \pi_1, \tau_1) \]

[I\text{LET}_m] \[ A \mid\| e_1 : \tau_1 \mid\| e_2 : \tau_2 \mid A \]
\[ \text{if} \ \{ X_i \} = \text{var}(t) \]
\[ \wedge \pi = \text{mgu}(\tau_1 \pi_1, \tau_1) \]

[I\text{LET}_p] \[ A \mid\| e_1 : \tau_1 \mid \]
\[ A \mid\| e_2 : \tau_2 \mid A \]
\[ \text{if} \ \{ X_i \} = \text{var}(t) \]
\[ \wedge \pi = \text{mgu}(\tau_1 \pi_1, \tau_1) \]

[I\text{LET}_p] \[ A \mid\| e : \tau \mid A \]
\[ A \mid\| e^* : \tau \mid A \]
\[ \text{if} \ \text{critVar}_{A*}(e) = \emptyset \]

Fig. 5. Inference rules
\(\lambda(snd'\ X).X : (\text{bool} \rightarrow \text{bool}) \rightarrow \text{int}\). It is clear that there is not a substitution more general than \([\alpha/\text{bool}]\) and \([\alpha/\text{int}]\) which makes possible a type derivation for \(\lambda(snd'\ X).X\). The only substitution more general than these two will be \([\alpha/\beta]\) (for some \(\beta\)), converting \(X\) in a critical variable.

In spite of this, we will see that \(\ldeq\cdot\) is still able to find the most general substitution when it exists. To formalize that, we will use the notion of \(\Pi_{A,e}^\ast\), which denotes the set collecting all type substitution \(\pi\) such that \(A\pi\) gives some type to \(e\).

**Definition 4 (Typing substitutions of \(e\)).**
\[
\Pi_{A,e}^\ast = \{ \pi \in T\text{Subst} \mid \exists \tau \in T\text{ype}. A\pi \vdash \cdot \ e : \tau \}
\]

Now we are ready to formulate our result regarding the maximality of \(\ldeq\cdot\).

**Theorem 6 (Maximality of \(\ldeq\cdot\)).**
\[
a) \Pi_{A,e}^\ast \text{ has a maximum element } \iff \exists \pi_g, \pi_g \in T\text{ype}. A \ldeq \cdot \ e : \tau_0 | \pi_g.
b) \text{ If } A\pi' \ldeq \cdot \ e : \tau' \text{ and } A \ldeq \cdot \ e : \tau | \pi \text{ then exists a type substitution } \pi'' \text{ such that } A\pi' = A\pi\pi'' \text{ and } \tau' = \tau\pi''.
\]

5 Type inference for programs

In the functional programming setting, type inference does not need to distinguish between programs and expressions, because the program can be incorporated in the expression by means of let expressions and \(\lambda\)-abstractions. This way, the results given for expressions are also valid for programs. But in our framework it is different, because our semantics (let-rewriting) does not support \(\lambda\)-abstractions and our let expressions do not define new functions but only perform pattern matching. Thereby in our case we need to provide an explicit method for inferring the types of a whole program. By doing so, we will also provide a specification closer to implementation.

The type inference procedure for a program takes a set of assumptions \(A\) and a program \(P\) and returns a type substitution \(\pi\). The set \(A\) must contain assumptions for all the symbols in the program, even for the functions defined in \(P\). We want to reflect the fact that in practice some defined functions may come with an explicit type declaration. Indeed this is a frequent way of documenting a program. Furthermore, type declarations are sometimes a real need, for instance if we want the language to support polymorphic recursion [16, 9]. Therefore, for some of the functions –those for which we want to infer types– the assumption will be simply a fresh type variable, to be instantiated by the inference process. For the rest, the assumption will be a closed type-scheme, to be checked by the procedure.

**Definition 5 (Type Inference of a Program).** The procedure \(B\) for type inference of a program \(\{\text{rule}_1, \ldots, \text{rule}_m\}\) is defined as:
\[
B(A, \{\text{rule}_1, \ldots, \text{rule}_m\}) = \pi, \text{ if }
\]
1. \( A \equiv \bullet (\varphi(\text{rule}_1), \ldots, \varphi(\text{rule}_m)) : (\tau_1, \ldots, \tau_m) | \pi. \)

2. Let \( f^1 \ldots f^k \) be the function symbols of the rules \( \text{rule}_i \) in \( P \) such that \( A(f^i) \) is a closed type-scheme, and \( \tau^i \) the type obtained for \( \text{rule}_i \) in step 1. Then \( \tau^i \) must be a variant of \( A(f^i) \).

\( \varphi \) is a transformation from rules to expressions defined as:

\[
\varphi(f \ t_1 \ldots t_n \to e) = \text{pair} \ \lambda t_1 \ldots \lambda t_n. e \ f
\]

where \( \text{pair} \) is the usual tuple constructor, with type \( \text{pair} : \forall \alpha_i. \alpha_1 \to \ldots \alpha_m \to (\alpha_1, \ldots, \alpha_m) \); and \( \text{pair} \) is a special constructor of tuples of two elements of the same type, with type \( \text{pair} : \forall \alpha. \alpha \to \alpha \to \alpha. \)

The procedure \( B \) has two important properties. It is sound: if the procedure \( B \) finds a substitution \( \pi \) then the program \( P \) is well-typed with respect to the assumptions \( A \pi \) (Th. 7). And second, if the procedure \( B \) succeeds it finds the most general typing substitution (Th. 8). It is not true in general that the existence of a well-typing substitution \( \pi' \) implies the existence of a most general one. A counterexample of this fact is very similar to Ex. 3.

**Theorem 7 (Soundness of \( B \)).** If \( B(A, P) = \pi \) then \( \text{wt}_A(\pi)(P). \)

**Theorem 8 (Maximality of \( B \)).** If \( \text{wt}_A(\pi')(P) \) and \( B(A, P) = \pi \) then \( \exists \pi'' \) such that \( A(\pi) = \pi'' \).

Notice that types inferred for the functions are simple types. In order to obtain type-schemes we need an extra step of generalization, as discussed in the next section.

### 5.1 Stratified Type Inference of a Program

It is known that splitting a program into blocks of mutually recursive functions and inferring the types in order may reduce the need of providing explicit typeschemes. This situation is shown in the next example.

**Example 4 (Program Inference vs Stratified Inference).**

\[
\begin{align*}
A & \equiv \{ \text{true} : \text{bool}, 0 : \text{int}, \text{id} : \alpha, f : \beta, g : \gamma \} \\
P & \equiv \{ \text{id} X \to X; f \to \text{id} \text{true}; g \to \text{id} \, 0 \} \\
P_1 & \equiv \{ \text{id} X \to X \}, P_2 \equiv \{ f \to \text{id} \text{true} \}, P_3 \equiv \{ g \to \text{id} \, 0 \}
\end{align*}
\]

An attempt to apply the procedure \( B \) to infer types for the whole program fails because it is not possible for \( \text{id} \) to have types \( \text{bool} \to \text{bool} \) and \( \text{int} \to \text{int} \) at the same time. We will need to provide explicitly the type-scheme for \( \text{id} : \forall \alpha. \alpha \to \alpha \) in order to the type inference to succeed, yielding types \( f : \text{bool} \to \text{bool} \) and \( g : \text{int} \to \text{int} \). But this is not necessary if we first infer types for \( P_1 \), obtaining \( \delta \to \delta \) for \( \text{id} \) which will be generalized to \( \forall \delta. \delta \to \delta \). With this assumption the type inference for both programs \( P_2 \) and \( P_3 \) will succeed with the expected types.

A general stratified inference procedure can be defined in terms of the basic inference \( B \). First, it calculates the graph of strongly connected components from
the dependency graph of the program, using e.g. Kosaraju or Tarjan’s algorithm [20]. Each strongly connected component will contain mutually dependent functions. Then it will infer types for every component (using \( B \)) in topological order, generalizing the obtained types before following with the next component.

Although stratified inference needs less explicit type-schemes, programs involving polymorphic recursion still require explicit type-schemes in order to infer their types.

6 Conclusions and Future Work

In this paper we have proposed a type system for functional logic languages based on Damas & Milner type system. As far as we know, prior to our work only [5] treats with technical detail a type system for functional logic programming. Our paper makes clear contributions when compared to [5]:

– By introducing the notion critical variables, we are more liberal in the treatment of opaque variables, but still preserving the essential property of subject reduction; moreover, this liberality extends also to data constructors, dropping the traditional restriction of transparency required to them. This is somehow similar to what happens with existential types [14] or generalized abstract datatypes [8], a connection that we plan to further investigate in the future.
– Our type system considers local pattern bindings and \( \lambda \)-abstractions (also with patterns), that were missing in [5]. In addition to that, we have made a rather exhaustive analysis and formalization of different possibilities for polymorphism in local bindings.
– Subject reduction was proved in [5] wrt. a narrowing calculus. Here we do it wrt. an small-step operational semantics closer to real computations.
– In [5] programs came with explicit type declarations. Here we provide algorithms for inferring types for programs without such declarations that can became part of the type stage of a FL compiler.

We have in mind several lines for future work. As an immediate task we plan to implement and integrate the stratified type inference into the TOY [11] compiler. Apart from the relation to existential types mentioned above, we are interested in other known extensions of type system, like type classes or generic programming. We also want to generalize the subject reduction property to narrowing, using let narrowing reductions of [10], and taking into account known problems [5, 1] in the interaction of HO narrowing and types. Handling extra variables (variables occurring only in right hand sides of rules) is another challenge from the viewpoint of types.

References


