Advances in type systems for Functional-Logic Programming
(Work in progress)

Francisco López-Fraguas\textsuperscript{1},
Enrique Martin-Martin\textsuperscript{2} and Juan Rodríguez-Hortalá\textsuperscript{3}

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Madrid, Spain

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. However, functional-logic languages have some problematic features not taken under consideration by standard systems. In particular, 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 a HO-let-rewriting, a recently proposed operational semantics for HO functional logic programs.

Keywords: Functional logic programming, Type systems, Opaque patterns

1 Introduction
Type systems for programming languages are an active area of research\textsuperscript{[16]}, no matters which paradigm one considers. In the case of functional programming, most type systems have arisen as extensions of Damas & Milner’s\textsuperscript{[4]}, for its remarkable simplicity and good properties (decidability, existence of principal types, possibility of type inference). Functional logic languages\textsuperscript{[11,8,7]}, 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

\textsuperscript{*} This work has been partially supported by the Spanish projects TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407 and UCM-BSCH-GR58/08-910502.
\textsuperscript{1} Email: fraguas@sip.ucm.es
\textsuperscript{2} Email: emartim@fdi.ucm.es
\textsuperscript{3} Email: juanrh@fdi.ucm.es

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
However, if types are not only decoration but are to provide safety, one should be sure that the adopted system has indeed good properties for the language under consideration, or at least for a well determined fragment. In this paper we tackle one aspect of existing FLP systems that are problematic or not well covered by standard Damas & Milner systems: the presence of so called HO patterns in programs. This is an expressive feature for which a sensible semantics exists [5]; however, unrestricted use of HO patterns leads to type unsafety, as recalled below.

The rest of the paper is organized as follows. The next subsection further introduce the mentioned aspect. Sect. 2 contains some preliminaries about FL programs and types. In Sect. 3 we expose the type system and prove its soundness wrt. let rewriting semantics. Sect. 4 contains a type inference relation, which let us find the most general type of expressions. Finally, Sect. 5 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 [6] that unrestricted use of HO patterns leads to loss of expected property of subject reduction (i.e., evaluation does not change types), an essential property for a type system. The following is a crisp example of the problem.

Example 1.1 (Polymorphic Casting [2]) Consider the program consisting of the rules snd X Y → Y, and true X → X, and false X → false, with the usual types inferred by a classical Damas & Milner algorithm. Then we can write the functions co (snd X) → X and cast X → co (snd X), whose inferred types will be ∀α.∀β.(α → α) → β and ∀α.∀β.α → β respectively. It is clear that and (cast 0) true is well-typed, because cast 0 has type bool (in fact it has any type), but if we reduce the expression using the rule of cast the resulting expression and 0 true is ill-typed.

The problem arises when dealing with HO patterns, because unlike FO patterns, knowing the type of a pattern does not always permit us to know the type of its subpatterns. In the previous example the cause is function co, because its pattern 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 the result of the function co.

In [6] 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 [6], prohibiting only those patterns having critical variables.
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 $Pat \ni t := X | c t_1 \ldots t_n \ (n \leq m) | f t_1 \ldots t_n \ (n < m)$ and the set of expressions $Exp \ni e := X | c | f | e_1 e_2 | \lambda t.e | \text{let } t = e_1 \text{ in } e_2$ where $c \in DC^m$ and $f \in FS^m$. We split the set of patterns into two: first order patterns $FOPat \ni \text{fot} := X | c t_1 \ldots t_n$ where $c \in DC^m$, and Higher order patterns $HOPat = Pat \setminus FOPat$. Expressions $e_1 \ldots e_m$ are called junk if $h \in CS^n$ and $m > n$, and active if $h \in FS^n$ and $m \geq n$. $FV(e)$ 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 $e_2$: $FV(\text{let } t = e_1 \text{ in } e_2) = FV(e_1) \cup (FV(e_2) \setminus FV(t))$. 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 $PRule \ni r := f t_1 \ldots t_n \rightarrow e \ (n \geq 0)$ where the set of patterns $T_i$ 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 $Prog \ni P := \{r_1, \ldots, r_n\} (n \geq 0)$.

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 then $\text{SType} \ni \tau := \alpha \mid \tau_1 \rightarrow \tau_2 \mid C \tau_1 \ldots \tau_n \ (C \in TC^n)$. Based on simple types we define the set of type-schemes as $\text{TScheme} \ni \sigma := \tau \mid \varnothing$. The set of free type variables (FTV) of a simple type $\tau$ is $\text{var}(\tau)$, and for type-schemes $\text{FTV}((\forall \alpha_i. \tau)) = \text{FTV}(\tau) \setminus \{\alpha_i\}$. A type-scheme $\forall \alpha_i. \tau \rightarrow \tau$ is transparent if $\text{FTV}(\tau_i) \subseteq \text{FTV}(\tau)$. A set of assumptions $\mathcal{A}$ is $\{s_i : \sigma_i\}$, where $s_i \in DC \cup FS \cup DV$. Notice that the transparency of type-schemes for data constructors 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 $\pi \in TSubst$ is a finite mapping from type variables to simple types $[\alpha_i/\tau_i]$. For sets of assumptions $\text{FTV}(\{s_i : \sigma_i\}) = \bigcup_i \text{FTV}(\sigma_i)$. We will say a type-scheme $\sigma$ is closed if $\text{FTV}(\sigma) = \emptyset$. Application of type substitutions to simple types is the natural, 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 $\sigma$ satisfying $\forall \alpha_i. \tau$ if $\tau' = \tau[\alpha_i/\tau_i]$ for some $\pi$, and we write it $\sigma \triangleright \tau'$. We extend $\triangleright$ to a relation between type-schemes by saying that $\sigma \triangleright \sigma'$ if every simple type such that $\sigma \triangleright \tau$ is also a generic instance of $\sigma$. Then $\forall \alpha_i. \tau \triangleright \forall \beta_i. \tau[\alpha_i/\beta_i]$ if $\{\beta_i\} \cap \text{FTV}(\forall \alpha_i. \tau) = \emptyset$ [3]. Finally, $\tau'$ is a variant of $\sigma$ satisfying $\forall \alpha_i. \tau$ ($\sigma \triangleright_{\text{var}} \tau'$) if $\tau' = \tau[\alpha_i/\beta_i]$ and $\beta_i$ are fresh type variables.

3 Type derivation

We propose a type system based in Damas & Milner's [4], but with some differences. We have divided our type system so that the task of giving a regular Damas & Milner
type and the task of checking critical variables are kept separated. To do that we have defined two different type relations: $\vdash$ and $\vdash^\ast$.

<table>
<thead>
<tr>
<th>ID</th>
<th>$\mathcal{A} \vdash s : \tau$ if $s \in DC \cup FS \cup DV$ and $(s : \sigma) \in \mathcal{A}$ and $\sigma \triangleright \tau$</th>
</tr>
</thead>
</table>
| APP      | $\mathcal{A} \vdash e_1 : \tau_1 \rightarrow \tau$  
$\mathcal{A} \vdash e_2 : \tau_1$                          |
| [A]      | $\mathcal{A} \vdash (X_i : \tau_i) \vdash t : \tau_2$ if $\{X_i\} = \text{var}(t)$                               |
|          | $\mathcal{A} \vdash \lambda e : \tau_1 \rightarrow \tau$                                                                     |
| LET      | $\mathcal{A} \vdash e_1 : \tau_1$  
$\mathcal{A} \vdash (X_i : \tau_i) \vdash e_2 : \tau_2$ if $\{X_i\} = \text{var}(t)$ |
|          | $\mathcal{A} \vdash \text{let } t = e_1 \text{ in } e_2 : \tau_2$                                                        |
| [P]      | $\mathcal{A} \vdash e : \tau$ if $\text{critVar}_\mathcal{A}(e) = \emptyset$                                                |

Fig. 1. Rules of type system

$\vdash$ is the basic typing relation (upper part of Fig. 1). It is like the classical Damas & Milner’s type system but extended to handle 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. Let expressions are considered monomorphically, i.e., the variables in the pattern of a let expression have the same type in all the occurrences in $e_2$.

The $\vdash^\ast$ relation (lower part of Fig. 1) 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 3.1 (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 $X_i \in \overline{X_i} = \text{var}(t)$ is opaque wrt. $\mathcal{A}$ iff $\exists \tau_i, \tau$ s.t. $\mathcal{A} \vdash (X_i : \tau_i) \vdash t : \tau$ and $\text{FTV}(\tau_i) \not\subseteq \text{FTV}(\tau)$.

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. 3.2 provides a more operational characterization of opacity that exploits the close relationship between $\vdash$ an type inference $\models$ presented in Sect. 4.

**Proposition 3.2** $X_i \in \overline{X_i} = \text{var}(t)$ is opaque wrt. $\mathcal{A}$ iff $\mathcal{A} \models \overline{X_i : \alpha_i} \models t : \tau_g | \pi_g$ and $\text{FTV}(\alpha_i, \pi_g) \not\subseteq \text{FTV}(\tau_g)$.

We write $\text{opaqueVar}_\mathcal{A}(t)$ for set of opaque variables of $t$ wrt. $\mathcal{A}$. Now, we can define the critical variables of an expression $e$ wrt. $\mathcal{A}$ as those variables that, being opaque in a let or lambda pattern of $e$, are indeed used in $e$. Formally:
Definition 3.3 (Critical variables of \( e \) wrt. \( A \))

\[
\text{critVar}_A(s) = \emptyset \quad \text{if} \ s \in \text{DC} \cup \text{FS} \cup \text{DV}
\]

\[
\text{critVar}_A(e_1 \cdot e_2) = \text{critVar}_A(e_1) \cup \text{critVar}_A(e_2)
\]

\[
\text{critVar}_A(\lambda t.e) = (\text{opaqueVar}_A(t) \cap \text{FV}(e)) \cup \text{critVar}_A(e)
\]

\[
\text{critVar}_A(\text{let } t = e_1 \text{ in } e_2) = (\text{opaqueVar}_A(t) \cap \text{FV}(e_2)) \cup \text{critVar}_A(e_1) \cup \text{critVar}_A(e_2)
\]

Notice that the if we write the function \( co \) of Ex. 1.1 as \( \lambda (\text{snd } X).X \), it is well-typed wrt. \( \vdash \) using the usual type for \( \text{snd} \). However it is ill-typed wrt. \( \vdash \bullet \) since \( X \) is an opaque variable in \( \text{snd } X \) and it occurs in the body, so it is critical.

The typing relation \( \vdash \bullet \) 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 \( \vdash \bullet \) 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 \texttt{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 \( \vdash \bullet \).

Theorem 3.4 (Properties of the typing relations)

\( a) \) If \( A \vdash ? e : \tau \) then \( A\pi \vdash ? e : \tau\pi \), for any \( \pi \in \mathcal{T}\text{Subst} \).

\( b) \) Let \( s \in \text{DC} \cup \text{FS} \cup \text{DV} \) be a symbol not occurring in \( e \). Then \( A \vdash ? e : \tau \iff A \oplus \{s : \sigma_s\} \vdash ? e : \tau \).

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

\( d) \) If \( A \oplus \{s : \sigma\} \vdash e : \tau \) and \( \sigma' \succ \sigma \), then \( A \oplus \{s : \sigma'\} \vdash e : \tau \).

Part \( a) \) states the closure under substitutions of the type derivations: if we have a correct type derivation, from an instance of the assumptions we could build a valid type derivation to an instance of the obtained type. \( b) \) shows that if there is a symbol which does not appear in an expression, we can add or delete any assumption over it from the set of assumptions and the type derivation will remain valid. Therefore in order to build a type derivation for an expression we only need to have assumptions for the symbols in the expression. \( c) \) states that if have a type derivation, we can substitute a variable for 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. This is not true wrt. the typing relation \( \vdash \bullet \) because a more general type can introduce opacity. For example the variable \( X \) is opaque in \( \text{snd } X \) with the usual type for \( \text{snd} \), but with a more specific type such as \( \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \) 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 3.5 (Well-typed program)** A program rule \( f \ t_1 \ldots t_n \rightarrow e \) is well-typed wrt. \( \mathcal{A} \) if \( \mathcal{A} \vdash \bullet \lambda t_1 \ldots \lambda t_n . e : \tau \) and \( \tau \) is a variant of \( \mathcal{A}(f) \). A program \( \mathcal{P} \) is well-typed wrt. \( \mathcal{A} \) if all its rules are well-typed wrt. \( \mathcal{A} \). If \( \mathcal{P} \) is well-typed wrt. \( \mathcal{A} \) we write \( \text{wt}_{\mathcal{A}}(\mathcal{P}) \).

Notice the use of the extended typing relation \( \vdash \bullet \) in the previous definition. This is essential, as we will explain later. Returning to Ex. 1.1, we can see that the program will not be well-typed because of the rule \( \text{co} (\text{snd } X) \rightarrow X \), since \( \lambda (\text{snd } X) . X \) will be ill-typed wrt. the usual type for \( \text{snd} \), 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 \( \mathcal{P} \equiv \{ f \text{ true } \rightarrow \text{ true } ; f \ 2 \rightarrow \text{ false } \} \) with the assumptions \( \mathcal{A} \equiv \{ f :: \forall \alpha . \alpha \rightarrow \text{bool} \} \). Besides, this restriction is implicitly considered in [6].

We also need a notion of evaluation. We have chosen *let-rewriting* [10], a recently proposed operational semantics for HO functional-logic programs. As can be seen in Fig. 2, the semantics 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 [15] that does not demand the matching if no variable of the pattern is needed, but forces the matching of the whole pattern if any variable is used. \( \lambda \)-abstractions have been omitted, since they are not supported by *let-rewriting*.

\[
\begin{align*}
\text{(Fapp)} & \quad f t_1 \theta \ldots t_n \theta \rightarrow^l r \theta, \quad \text{if } (f t_1 \ldots t_n \rightarrow r) \in \mathcal{P} \text{ and } \theta \in \mathcal{P}_{\text{Subst}} \\
\text{(LetIn)} & \quad e_1 e_2 \rightarrow^l \text{let } X = e_2 \text{ in } e_1 X, \quad \text{if } e_2 \text{ is an active expression, variable application, junk or let rooted expression, for } X \text{ fresh.} \\
\text{(Bind)} & \quad \text{let } X = t \text{ in } e \rightarrow^l e[X/t], \quad \text{if } t \in \text{Pat} \\
\text{(Elim)} & \quad \text{let } X = e_1 \text{ in } e_2 \rightarrow^l e_2, \quad \text{if } X \notin \text{FV}(e_2) \\
\text{(Flat)} & \quad \text{let } X = (\text{let } Y = e_1 \text{ in } e_2) \text{ in } e_3 \rightarrow^l \text{let } Y = e_1 \text{ in } (\text{let } X = e_2 \text{ in } e_3), \quad \text{if } Y \notin \text{FV}(e_3) \\
\text{(LetAp)} & \quad (\text{let } X = e_1 \text{ in } e_2) e_3 \rightarrow^l \text{let } X = e_1 \text{ in } e_2 e_3, \quad \text{if } X \notin \text{FV}(e_3) \\
\text{(Contx)} & \quad C[e] \rightarrow^l C[e'], \quad \text{if } C \neq [ ], e \rightarrow^l e' \text{ using any of the previous rules}
\end{align*}
\]

Fig. 2. Higher order let-rewriting relation \( \rightarrow^l \)

The proposed transformation preserves the types of the expressions, as it is
stated in Th. 3.6.

\[
\text{TRL}(s) = s, \text{ if } s \in DC \cup FS \cup DV \\
\text{TRL}(e_1 e_2) = \text{TRL}(e_1) \text{ TRL}(e_2) \\
\text{TRL}(\text{let } X = e_1 \text{ in } e_2) = \text{let } X = \text{TRL}(e_1) \text{ in } \text{TRL}(e_2) \\
\text{TRL}(\text{let } t = e_1 \text{ in } e_2) = \text{let } Y = \text{TRL}(e_1) \text{ in } \text{let } X_i = f_{X_i} Y \text{ in } \text{TRL}(e_2) \\
\text{for } \{X_i\} = \text{var}(t) \cap \text{var}(e_2), f_{X_i} \in FS^1 \text{ fresh defined by the rule } f_{X_i} t \rightarrow X_i, \\
Y \in DV \text{ fresh and } t \text{ a non variable pattern. }
\]

Th. 3.6 also states that the projection functions are well-typed. Then if we start from a well-typed program \(P\) wrt. \(\mathcal{A}\) and apply the transformation to all its rules, the program extended with the projections rules will be well-typed wrt. the extended assumptions: \(wt_{\mathcal{A} \oplus \mathcal{A}'}(P \uplus P')\). This result is straightforward, because \(\mathcal{A}'\) does not contain any assumption for a symbol in \(P\), so \(wt_{\mathcal{A}}(P)\) implies \(wt_{\mathcal{A} \oplus \mathcal{A}'}(P)\).

Th. 3.7 states the subject reduction property for one step of let-rewriting, but its extension to any number of steps is trivial. As we have said, let-rewriting does not support lambda expressions nor let expressions with compound patterns, so the subject reduction is only valid for the restricted set of expressions without these constructions.

**Theorem 3.6 (Type preservation of the let transformation)**

Let be \(\mathcal{A} \vdash^* e : \tau\) and \(P \equiv \{f_{X_i} : t_i \rightarrow X_i\}\) the rules of the projections functions needed to eliminate the let expressions with compound patterns in \(e\) according to Fig. 3. Let also \(\mathcal{A}'\) be the set of assumptions over that functions, defined as \(\mathcal{A}' \equiv \{f_{X_i} : \text{Gen}(\tau_{X_i}, \mathcal{A})\}\), where \(\mathcal{A} \vdash^* \lambda t_i. X_i : \tau_{X_i} | \pi_{X_i}\). Then \(\mathcal{A} \oplus \mathcal{A}' \vdash^* \text{TRL}(e) : \tau\) and \(wt_{\mathcal{A} \oplus \mathcal{A}'}(P)\).

Th. 3.6 also states that the projection functions are well-typed. Then if we start from a well-typed program \(P\) wrt. \(\mathcal{A}\) and apply the transformation to all its rules, the program extended with the projections rules will be well-typed wrt. the extended assumptions: \(wt_{\mathcal{A} \oplus \mathcal{A}'}(P \uplus P')\). This result is straightforward, because \(\mathcal{A}'\) does not contain any assumption for a symbol in \(P\), so \(wt_{\mathcal{A}}(P)\) implies \(wt_{\mathcal{A} \oplus \mathcal{A}'}(P)\).

Th. 3.7 states the subject reduction property for one step of let-rewriting, but its extension to any number of steps is trivial. As we have said, let-rewriting does not support lambda expressions nor let expressions with compound patterns, so the subject reduction is only valid for the restricted set of expressions without these constructions.

**Theorem 3.7 (Subject Reduction)**

If \(\mathcal{A} \vdash^* e : \tau\) and \(wt_{\mathcal{A}}(P)\) and \(P \vdash e \rightarrow^i e'\) then \(\mathcal{A} \vdash^* e' : \tau\).

It is essential that the definition of well-typed program relies on \(\vdash^*\), because it assures that the program rules do not have critical variables, and therefore the reduction of function calls will not create problems with the types. If this definition was based only in \(\vdash\) and did not check the critical variables then the subject reduction would not hold. A counterexample can be found in Ex. 1.1, where the program will be well-typed with this relaxed form of well-typeness but the subject reduction property will fail for and (cast 0) true because of the rule for co.

The proof of the subject reduction property is based on an important result about transparent variables: the well behavior of the transparent variables of a pattern when they are instantiated (Lemma 3.8). 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 changing the transparent variables by expressions with the same type. This is not guaranteed with opaque variables, and that is why we forbid their use in expressions.
Lemma 3.8
Assume \( A \oplus \{ X_i : \alpha_i \} \vdash t : \tau \), 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

The typing relation lacks some properties that prevents 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 \( \vdash^* \) 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. And finally, the typing relation needs all the assumptions for the symbols in order to work. To solve these problems, type systems usually are accompanied with a type inference algorithm which returns a valid type for an expression and also establishes the types for some symbols in the expression.

In this work we have given the type inference in Fig. 4 a relational style to show the similarities with the typing relation. But in essence, these rules for inference represent an algorithm (as algorithm \( W \) [4,3]) 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 derive a type for \( e \).

\[
\begin{align*}
\text{[II\text{D}] } & \quad A \vdash s : \tau | t \quad \text{if } s \in DC \cup FS \cup VAR \land (\{ s : \sigma \} \in A \land \sigma \succ \text{var } \tau) \\
\text{[IA\text{PP}] } & \quad A \vdash e_1 : \tau_1 | \pi_1 \quad A \vdash e_2 : \tau_2 | \pi_2 \\
& \quad A \vdash e_1 e_2 : \alpha | \pi_1 \pi_2 \quad \text{if } \alpha \text{ fresh type variable } \\
& \quad \land \pi = \text{mgu}(\tau_1 \pi_2, \tau_2 \rightarrow \alpha) \\
\text{[IA] } & \quad A \oplus \{ X_i : \alpha_i \} \vdash t : \tau_i | \pi_t \quad \text{if } \{ X_i \} = \text{var}(t) \land \alpha_i \text{ fresh type variables} \\
& \quad (A \oplus \{ X_i : \alpha_i \} \pi_t) \vdash e : \tau | \pi \\
& \quad A \vdash \lambda e \rightarrow t : \tau | \pi_\tau \quad \text{if } \\{ X_i \} = \text{var}(t) \\
\text{[LET] } & \quad A \oplus \{ X_i : \alpha_i \} \vdash t : \tau_i | \pi_t \quad \text{if } \{ X_i \} = \text{var}(t) \land \alpha_i \text{ fresh type variables} \\
& \quad (A \oplus \{ X_i : \alpha_i \} \pi_t) \vdash e_1 \rightarrow t \quad \land \pi = \text{mgu}(\tau_1 \pi_1, \tau_1) \\
& \quad A \vdash \text{let } t = e_1 \text{ in } e_2 : \tau \quad \text{if } \text{critVar}_{A\pi}(e) = \emptyset \\
\text{[IP] } & \quad A \vdash e : \tau | \pi \\
& \quad A \vdash^* e : \tau | \pi \quad \text{if } \text{critVar}_{A\pi}(e) = \emptyset
\end{align*}
\]

Fig. 4. Inference rules

Th. 4.1 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 4.1 (Soundness of \( \vdash^* \)) \( A \vdash^* e : \tau | \pi \implies A\pi \vdash e : \tau \)
Th. 4.2 expresses the completeness of the inference. If we can derive a type for an expression applying a substitution to the assumptions, then inference will succeed and will find a most general type and substitution.

**Theorem 4.2 (Completeness of ⊬ wrt. ⊢)**

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. 4.2 cannot be obtained for $\vdash •$ because of critical variables, as Ex. 4.3 shows.

**Example 4.3 (Inexistence of a more general substitution)**

Let $A$ be the set of assumptions $\{ \text{snd}' : \alpha \rightarrow \text{bool} \rightarrow \text{bool} \}$ and consider the following two valid derivations $D_1 \equiv A[\alpha/\text{bool}] \vdash • \lambda (\text{snd}' X).X : (\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool}$ and $D_2 \equiv A[\alpha/\text{int}] \vdash • \lambda (\text{snd}' X).X : (\text{bool} \rightarrow \text{bool}) \rightarrow \text{int}$. 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 $\lambda (\text{snd}' X).X$. 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, Th. 4.5 shows that inference $\vdash •$ will succeed and find a more general substitution and type in the case that a more general substitution exists. To formalize that, we will use the notion of $\Pi_{A,e}^•$, which denotes the set collecting all type substitution $\pi$ such that $A\pi$ gives some type to $e$.

**Definition 4.4 Typing substitutions of $e$**

$\Pi_{A,e}^• = \{ \pi \in T\text{Subst} \mid \exists \tau \in ST\text{ype}. A\pi \vdash • e : \tau \}$

Now we are ready to formulate our result regarding the maximality of $\vdash •$.

**Theorem 4.5 (Maximality of $\vdash •$)**

a) $\Pi_{A,e}^•$ has a maximum element $\iff \exists \tau_g \in ST\text{ype}, \pi_g \in T\text{Subst}. A \vdash • e : \tau|\pi_g$.

b) If $A\pi' \vdash • e : \tau'$ and $A \vdash • e : \tau|\pi$ then exists a type substitution $\pi''$ such that $A\pi' = A\pi\pi''$ and $\tau' = \tau\pi''$.

5 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 [6] treats with technical detail a type system for functional logic programming. Our paper makes clear contributions when compared to [6]:

- By introducing 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 constructor, dropping in this way the traditional restriction of transparency required to data constructors. This is somehow similar to what happens with existential types [13] or generalized algebraic datatypes [9], 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 [6].
• Subject reduction was proved in [6] wrt. a narrowing calculus. Here we do it wrt. an small-step operational semantics closer to real computations.

We have in mind several lines for future work: apart from the relation to existential types mentioned above, we are interested in other known extensions of type systems, 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 [6,1] 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