Abstract

Functional logic programming (FLP) is a paradigm that comes from the integration of lazy functional programming and logic programming. Although most FLP systems use static typing by means of a direct adaptation of Damas-Milner type system, it is well-known that some FLP features like higher-order patterns or the equality operator lead to so-called opacity situations that are not properly handled by Damas-Milner type system, thus leading to the loss of type preservation. Previous works have addressed this problem either directly forbidding those HO patterns that are opaque or restricting its use. In this paper we propose a new approach that is based on eliminating the unintended opacity created by HO patterns and the equality operator by extending the expressiveness of the type language with decorations in the arrows of the functional types. We study diverse possibilities, which differ in the amount of information included in the decorations. The obtained type systems have different properties and expressiveness, but each of them recovers type preservation from simple extensions of Damas-Milner.

Categories and Subject Descriptors E.3.3 [Logics and meanings of programs]: Studies of Program Constructs—Type Structure; D.3.1 [Programming Languages]: Language Classifications—Multiparadigm languages; D.3.2 [Programming Languages]: Formal Definitions and Theory

General Terms Theory, Languages, Design

Keywords Functional-logic programming, type systems, higher-order patterns, opacity

1. Introduction

Functional logic programming (FLP) is a paradigm that comes from the integration of the main features of lazy functional programming and logic programming. Hence, modern FLP languages like Toy or Curry can be roughly described as a variant of Haskell with some modifications and extensions at the semantic level. First of all, overlapping rules are not handled in a first fit approach like in Haskell, but they are tried in order using a backtracking mechanism in the line of Prolog. This leads to the definition of so-called non-deterministic functions, which may return more than one result for the same input. This combination of non-determinism and lazy evaluation gives rise to several semantic options, among which call-time choice semantics is the option adopted by most modern FLP implementations. Call-time choice corresponds to call-by-need parameter passing in the sense that different occurrences of the same variable in the body of a program rule share the same value. To illustrate this point let us consider the FLP program {com \(\rightarrow\) \(z\), \(\text{com} \rightarrow s\ z\), \(\text{pair} X \rightarrow (X, X)\)} where \(z\) and \(s\) stand for the data constructors for Peano numbers. Under a call-time choice semantics the values \((z, z)\) and \((s, s)\) are correct for the expression \(\text{pair com}\), but the values \((z, s z)\) and \((s, z s)\) are incorrect because the occurrences of \(X\) in \((X, X)\) must share the same value.

As a consequence of non-determinism, the notion of equality is also revised in FLP languages. Given two expressions \(e_1\) and \(e_2\), several interpretations for their equality are possible. For example, we could ask for both expressions to have the same set of values, which is not very practical in a lazy language, as those sets can easily be infinite. The criterion adopted in modern FLP languages corresponds to the notion of joinability, so two expressions \(e_1\) and \(e_2\) are joinable, written \(e_1 \bowtie e_2\), iff they can be reduced to the same value.

There are different approaches to functional-logic programming, but in this work we will use the same approach as Toy, which is based on the HO-CRWL logic. In FLP, due to the combination of higher order features, call-time choice and non-determinism, expressions that are extensionally equal—i.e., that have the same behavior when applied to the same arguments—can produce different values when placed in the same context. The HO-CRWL logic follows an intensional approach that semantically distinguishes function symbols for extensionally equivalent functions whenever they are syntactically different. This intensionality leads to another important feature of the functional-logic language Toy, namely higher-order patterns. These patterns are composed by partial applications of function or constructor symbols to other patterns, thus generalizing the notion of patterns that can appear in left-hand sides of rules in Haskell. By using HO patterns, functions are not treated as black boxes anymore but can be distinguished by matching. For example, programmers can define different sorting functions for lists, e.g. \(\text{quicksort}\) and \(\text{permutsort}\). They correspond to the same extensional sorting function, however, \(\text{quicksort}\) and \(\text{permutsort}\) are two different intensional descriptions that can be distinguished in the left-hand side of a rule: \{\(\text{tractable quicksort} \rightarrow \text{true}\), \(\text{tractable permutsort} \rightarrow \text{false}\)\}. Thanks to this ability to view functional expressions as data that can appear in left-hand sides of rules, HO patterns have been proved to be a useful and expressive feature. With the aim of providing standard tools for reasoning in FLP—like type-based rea-

---

\[1\] We will use an applicative syntax similar to Haskell syntax but employing uppercase for variables and lowercase for constructor and function symbols.

\[2\] HO-CRWL stands for Higher-Order Constructor Based Rewriting Logic, the higher order extension of CRWL.
soning via free theorems\footnote{http://www.informatik.uni-kiel.de/~pakcs/}—a new denotational semantics has been recently proposed\footnote{http://danae.uni-muenster.de/~lux/curry/}. This semantics, which does not consider HO patterns because it is proposed for the Curry language, is more “abstract” than HO-CRWL, considering as semantically equal functional expressions that are different in HO-CRWL like id and map id. However, in this work we have considered the HO-CRWL approach because it a well-established semantics for FLP\cite{15} and it is at the core of the language Toy.

Regarding types, most FLP systems like Toy or the implementations of Curry use static typing by means of a direct adaptation of Damas-Milner type system\cite{10}, where the equality operator is added as a primitive of the language with type \( \triangleright \triangleright : \forall \alpha.\alpha \rightarrow \alpha \rightarrow \text{bool} \). The reason for that is twofold. First of all, overloading support by means of type classes is still in an experimental phase\cite{24,25}. But, more importantly, \( \triangleright \triangleright \) cannot be defined as an ordinary function because it has to compare expressions with variables, which contrary to Haskell and other functional languages, are valid run-time expressions. Therefore, no program rule can express that \( X \triangleright \triangleright X \) should be reduced to \text{true} as establishes the notion of joinability of HO-CRWL semantics\cite{12}—e.g. the rule \( V \triangleright \triangleright V \rightarrow \text{true} \) is not valid as it is not left-linear, a usual requirement in lazy functional languages.

Nevertheless, it is well-known\cite{13} that some FLP features like the equality operator or the use of HO patterns lead to so-called opacity situations that are not properly handled by Damas-Milner typing, thus leading to the loss of type preservation. The following examples, which borrow some ideas from\cite{13,20}, illustrate these problems.

\textbf{Example 1.} Consider a function \text{snd} defined by the rule

\[
\text{snd} \quad X \rightarrow Y
\]

for which the classical Damas-Milner algorithm infers the type \( \text{snd} : \forall \alpha,\beta.\alpha \rightarrow \beta \rightarrow \beta \). The point is that in any partial application of \text{snd}, the type of its argument is not reflected in the type of the whole expression: for example \text{snd} \( z \) has type \( \beta \rightarrow \beta \), in which we cannot find the type \( \text{nat} \) that corresponds to \( z \).

This situation can be described using the terminology “opacity”\footnote{Admittedly, equality between higher-order expressions is not specified in the Curry Report\cite{14}, however, it is supported in the mentioned implementations.}, so a symbol is called opaque if the type of each of its arguments is not determined by the type of the application of the symbol to those arguments. This way the type of expressions placed in an opaque context is unknown. Opaque symbols are dangerous whose implementations are first-class citizens, i.e., they can be passed as function parameters or returned by functions. Although existential types maybe could be adapted to safely handle the opacity caused by HO patterns in program rules, it is not clear how they could be adapted to avoid opaque decomposition, as the equality operator is not defined by a set of program rules that can be accepted or rejected separately by the type system. Therefore, the dangerous expression \( \text{snd} \triangleright \triangleright \text{snd} \triangleright \triangleright \text{true} \) would still be a valid expression using an adaptation of existential types.

In the context of FLP, the seminal work\cite{13} already identified those unintended opacity situations, so opaque patterns are forbidden and type preservation is only granted for computations with no opaque decomposition steps, which is undecided. Some other works have been developed recently to allow safe uses of opaque HO patterns\cite{19,20}. They restrict the use of variables whose type has been hidden by opacity, employing techniques different from those used in existential types, obtaining type systems with diverse properties and possibilities for generic programming techniques. However, the use of the equality operator and the subsequent problem of opaque decomposition is not treated.

To the best of our knowledge there is no proposal for solving the problem of opaque decomposition. In this paper we propose a new
approach to overcome these problems that is based on eliminating the unintended opacity created by HO patterns and the equality operator, by proposing several different extensions of the type language. The resulting type systems are simple extensions of Damas-Milner typing that recover type preservation. The idea is, starting from the transparency hypothesis, to ensure transparency of patterns as an invariant during type inference of programs. The unexpected opacity might only appear in HO patterns, as it cannot be caused by partial applications of constructor symbols, because if a symbol is transparent then all its partial applications are transparent too. The problem lies in the partial applications of function symbols that appear in HO patterns: for each function of arity $n$ there are $n$ possible partial applications that conceptually correspond to $n$ constructor symbols. How do we fix this? We take inspiration from what we would do in Haskell for the declaration `data t B = sndc & B, that is rejected because the variable $A$ does not appear as an argument of the type constructor $t$, which imposes the opaque assumption $\text{sndc} : \forall \alpha, \beta. \alpha \rightarrow \beta \rightarrow t \beta$—which is very similar to the type for $\text{snd}$ in Example 1. We can easily fix this by adding $A$ as an additional argument for $t$, getting the transparent assumption $\text{sndc} : \forall \alpha, \beta. \alpha \rightarrow \beta \rightarrow t \alpha \beta$. As the type constructor for functions is the arrow $\rightarrow$, we can reproduce this solution for partial applications of functions by adding a new argument to $\rightarrow$ in particular one carrying the types of the arguments already applied to the function. As we will see in this paper $\forall \alpha, \beta. \alpha \rightarrow \beta \rightarrow t \beta$ is a valid type for the function $\text{snd}$ in Example 1 according to our type systems, thus clearing up the opacity of $\text{snd}$. As a consequence, the function $\text{cast}$ takes the type $\text{cast} : \forall \alpha, \beta. \alpha \rightarrow (\forall \gamma. \beta \rightarrow t \gamma)$, which corresponds to its actual behaviour, and the equality $\text{snd} z \bis \text{snd} \text{true}$ is rejected as $\beta \rightarrow (\forall (\text{nat}) \beta$ is different to $\beta \rightarrow (\forall (\text{bool}) \beta$. In fact the problem of opaque decomposition is avoided because there is no opacity at all—as they say, “dead dogs don’t bite”.

The rest of the paper is organized as follows. Section 2 contains some technical preliminaries and notions about expressions, types and the operational semantics used in this paper. In Section 3 we present a type system using an approach that decorates arrows with complete types. We also show how it eliminates opacity from programs if we assume the transparency hypothesis, and provide a complete type inference procedure for expressions. We also prove its soundness w.r.t. the semantics. On the negative side, it considers ill-typed some safe programs using HO patterns. To overcome this limitation, in Section 4 we present a modification of the previous type system that reduces the amount of information included in arrow decorations. The resulting type system still preserves types, however it lacks closure under type substitutions, which is an important property since it is used intensively in the inference algorithm for expressions. Mixing the ideas of the two previous type systems, in Section 5 we present a type systems that preserves types and is also closed under type substitutions, so type inference for expressions is possible. In Section 6 we compare the presented type systems with previous proposals for FLP. Finally, Section 7 summarizes some conclusions and future work. The complete proofs of the results—as well as complementary material about type inference for programs—can be found in the extended version of this paper.

---

2. Preliminaries

2.1 Expressions and Programs

We assume a signature $\Sigma = CS \cup FS$, where $CS$ and $FS$ are two disjoint sets of data constructors $c$ and function symbols $f$ respectively. Each symbol $h \in \Sigma$ has an associated arity $\text{ar}(h)$. We also assume a denumerable set $DV$ of data variables $X$. Figure 1 shows the syntax of expressions $e \in \text{Expr}$—our notion of value—and expressions $e \in \text{Expr}$ use the notation $\sigma_0$ for a sequence of $n$ syntactic objects $o_1, \ldots, o_n$ of $\Sigma$, where $o_i$ refers to the $i$th element of the sequence. If the number of elements is not relevant, we write simply $\sigma$. We split the set of patterns into two: first-order patterns $\text{FOPat} \ni \text{fot} := X | e \text{for} t$, where $\text{ar}(e) = n$, and higher-order patterns $\text{HOPat} = \text{Pat} \setminus \text{FOPat}$. We also distinguish different classes of expressions: $X \tau_n (n > 0)$ is a variable application, $c \tau_n$ is a junk expression when $n > \text{ar}(c)$ and $f \tau_n$ is an active expression when $n > \text{ar}(f)$. The set of variables—$\text{var}(e)$—and free variables—$\text{fv}(e)$—are defined in the usual way. Note that our let-expressions are not recursive, so $\text{fv}(\text{let } X = e_1 \text{ in } e_2) = \text{fv}(e_1) \cup \text{fv}(e_2) \setminus \{X\}$.

A program rule $R$ is defined as $f \tau_n \rightarrow e$ where $\text{ar}(f) = n$ and $\tau_n$ is linear, i.e., every variable occurs only once in all the programs. Program rules also verify that $\text{fv}(e) \subseteq \bigcup_{n=1}^{\text{ar}(e)} \text{fv}(t_i)$, so extra variables are not considered in this work. Programs $P$ are sets of rules $\{R_1, \ldots, R_n\}$. A one-hole context is defined as $C := [] | C \ c \ c \ C \ | \ let \ X \ = \ C \ in \ e | let \ X = e \ in C$, and its application to an expression—$C[e]$—is defined in the usual way. The set of bound variables of a context—$b(C)$—contains those variables bound by a let-expression in the context, and it is defined as $b(C[\{\}]) = \emptyset$, $b(C \ c \ c \ c) = b(C \ c) \cup b(C)$, $b(\text{let } X = C \ in \ e \ c) = b(C \ c) \cup b(\text{let } X = e \ in C) = \{X\} \cup b(C \ c)$. A data substitution $\theta \in \mathcal{PSub}$ is a finite mapping from data variables to patterns $\{X \mapsto t\}$. The domain and variable range of a data substitution are defined as $\text{dom}(\theta) = \{X \in DV \mid X \not\in \theta\}$ and $\text{var}(\theta) = \bigcup_{X \in \text{dom}(\theta)} \text{fv}(X \theta)$, respectively. Application of data substitutions is defined in the natural way.

2.2 Types

We assume a denumerable set $TV$ of type variables $\alpha$ and a countable alphabet $TC$ of type constructors $C$. Each type constructor $C$ has an associated arity $\text{ar}(C)$. The syntax of simple types $\tau$ and type-schemes $\sigma$ appears in Figure 1. The main novelty regarding the usual syntax of simple types is that arrows in functional types must be decorated with a type variable or a tuple—possibly empty—of types. For instance, $\text{bool} \rightarrow \text{bool}$ and $\text{int} \rightarrow \text{nat} \ [\text{bool}]$ are syntactically invalid types, while $\text{bool} \rightarrow (\text{bool} \times \text{nat}) [\text{bool}]$ are correct ones. These type decorations $\rho$ in the arrows are a crucial ingredient for removing opacity from higher-order patterns as

\[ \text{Symbol} \quad \sigma := X | f | \alpha \]
\[ \text{Pattern} \quad t := X | e t_0 (n \leq \text{ar}(e)) \]
\[ \text{Expression} \quad e := X | e | f | e_1 e_2 \]
\[ \text{Simple type} \quad \tau := \alpha | (\forall \gamma. \alpha) \]
\[ \text{Type decoration} \quad \rho := \alpha \]
\[ \text{Type scheme} \quad \sigma := \forall \gamma. \tau (n \geq 0) \]

---

For the sake of notation homogeneity we employ here a modification of Haskell’s syntax that uses uppercase for variables and lowercase for constructors.
they are used to store the types of the arguments already applied in partial applications of functions, as we will see in Sections 4 and 5. The set of free type variables (ftv) of a simple type τ is defined in the usual way except for functional types, where it is defined as: ftv(∀σ.τ) = ftv(τ) \setminus \{σ\}. For type-schemes, ftv(∀σ.τ) = ftv(τ) \setminus \{σ\}. A type-scheme σ ⊢ ∀τ₁...τₙ → τₖ is called k-transparent if ftv(τ₁) \setminus \{τ₁\} ⊆ ftv(τ'), and closed if ftv(τ') = ∅. Notice that k-transparency implies j-transparency for any j < k, and every type-scheme is trivially 0-transparent.

A set of assumptions A is a set {σ₁ = τ₁} relating type-schemes to symbols. If (s : σ) ∈ A we write A(s) = σ. Data constructors and function symbols appearing in A must be transparent, i.e., A(c) must be k-transparent if k_c = ar(c), and A(f) must be k_f-transparent if k_f = ar(f) − 1. For function symbols we do not require transparency for their total application because function symbols can only appear partially applied in patterns. This transparency hypothesis is essential to guarantee type preservation with equality, as we will see in Section 5. The transparency requirement is common over data constructors, and forbids the use of existential types [18], but it might seem too tight for functions, at first sight. Nevertheless, as we will see in Sections 4 and 5, when transparent assumptions are used our type systems only infer transparent types for functions. As in practice type inference for programs starts from a transparent set of assumptions for constructors—obtained from data type declarations—and infers the types for the functions in an order according to the dependencies in the call-graph, we may conclude that thanks to the “transparency invariant” that requirement is not limiting and always holds in practice. The set of free type variables of a set of assumptions is defined as ftv({σ₁ = τ₁}) = \bigcup_{i=1}^{n} ftv(σ_i). The union of sets of assumptions is denoted by A ⊕ A' with the usual meaning: it contains the assumptions in A' as well as those in A for the symbols not appearing in A'.

A type substitution τ is a finite mapping from type variables to simple types [σ = τ], whose application is defined in the usual way. The domain dom(τ) and variable range ran(τ) are defined as for data substitutions. We use τ to denote the identity substitution. A simple type τ' is a generic instance of σ = ∀τ.τ if τ' = τ[α → τ'], for some τ', and we write σ ⊳ τ'. Finally, we say τ' is a variant of σ = ∀τ.τ (written σ ⊳ τ) if τ' = τ[α → β], where β are free type variables.

2.3 Operational Semantics

The operational semantics used in this work is based on let-rewriting [22]. This semantics, which is sound and complete w.r.t. HO-CRWL, is a notion of reduction step that expresses call-time choice by means of sharing subexpressions using let-bindings. In Figure 2 we have extended the original let-rewriting relation with two rules to cope with an equality operator (⊩) that corresponds to the notion of joinability [12]. This equality behaves like the strict equality operator of Curry, so two expressions are equal when they can be reduced to the same value. However, notice that this operational semantics supports equality between functional expressions—its rules handle equality between HO-patterns—in contrast with the specification of Curry [14], that only allows equalities for first-order expressions.

We assume that true and false are 0-arity constructor symbols, and ∧ is a binary function symbol defined with the rule true ∧ Y → Y. Regarding types, we assume that every set of assumptions contains: {true : bool, false : bool, (λ) : bool → ⊥} (LetIn) e₁ e₂ → e₁ let X = e₂ in e₁ X, if e₂ is junk, active, variable application or a let-expression; for X fresh.

(Bind) let X = t in e → e[X/t] (Elim) let X = e₁ in e₂ → e₂, if X \notin ftv(e₂) (Flat) let Y = (let Y = e₁ in e₂) in e₃ → e₁ let Y = e₁ in (let X = e₂ in e₃), if Y \notin ftv(e₃) (LetAp) (let X = e₁ in e₂) e₃ → e₁ let X = e₁ in e₂ e₃ if X \notin ftv(e₃)

(Fapp) f t₁...tn → r if (f t₁...tn → r) ∈ P (Join) s ≺ t → true, if s \in Pat (JoinP) (h τ₁) → (h τ₂) → t₁ (t₂ \Rightarrow t₃) ∧ ... ∧ (tₙ \Rightarrow t₀') if (h τ₁), (h τ₂) ∈ Pat and n > 0.

(Contx) C[e] → C'[e'], if C ≠ [], e → e' using any of the previous rules, and if the step is X ≺ X → true using (JoinS) then X \notin be(C)

Figure 2. Higher order let-rewriting relation with equality →

bool → γ]bool bool, (∞) : ∀α.α → γ₁ α → γ₁ (α) bool}. The first five rules (LetIn)–(LetAp) do not use the program and just change the textual representation of the term graph implied by the let-bindings in order to enable the application of program rules or the equality operator, but keeping the implied term graph untouched. The (Fapp) rule performs function application. The rule (JoinS) reduces the equality of one-symbol patterns to true. It is important to force them to be patterns, otherwise the rule could be incorrectly applied to the equality of 0-arity function symbols—coin or coin—without evaluating them. On the other hand, the rule (JoinP) reduces the equality of compound patterns by decomposition. This step is the cause of the opaque decomposition problem mentioned in Section 3. Note that the rules (JoinS) and (JoinP) represent the same behavior as joinability. Therefore an equality involving two patterns t₁ \Rightarrow t₂ will be evaluated to true iff both patterns are syntactically the same—including HO patterns. In addition to the new rules, (Contx) is modified to avoid the application of (JoinS) to bound variables, as bound variables do not correspond to totally defined values but to expressions whose evaluation is pending [22], and by definition joinability only holds for totally defined values [12]. Finally, we also assume that programs do not contain any rule for ∞, so (Fapp) cannot be used to evaluate an equality.

3. Fully decorated type system

In this section we present a type system that decorates the arrows of functional types with the complete information about the types of the previous arguments. Although this simple approach leads to a type system that recovers type preservation and has a sound complete type inference, we will show that it is not expressive enough and rejects some well-known programs using HO patterns. Sections 4 and 5 contain two different approaches trying to overcome this limitation.

We consider the type derivation relation A ⊢ e : τ in Figure 3 where A ⊢ e : τ means that the expression e has type τ under the assumptions A. Notice that, although they do not appear in the syntax presented in Figure 1 and they cannot appear in programs, Figure 3 includes a rule for \(\lambda\)-abstractions of expressions with the shape \(\lambda x.\)
This kind of expressions have been included only for typing purposes, as they simplify the definition of well-typed program rule: a rule \( f \Gamma \rightarrow e \) is well-typed whenever its associated \( \lambda \)-abstraction \( \lambda f. e \) matches the type for \( f \) (see Definition 2). The type derivation relation in Figure 3 is based on our previous type system for FLP [20], so it is similar to Damas-Milner [10] modified to be syntax-directed. The main novelty lies in the treatment of \( \lambda \)-abstractions, as the rule (\( \Lambda \)) now decorates each of the arrows in the type obtained for a chain of \( \lambda \)-abstractions, with the types of its previous arguments. For that, (\( \Lambda \)) uses the \text{adepth} and \text{anArgs} meta operators:

### Definition 1 (\( \lambda \)-depth and type decoration).

- \text{adepth}(s) = 0
- \text{adepth}(\text{let } X = e_1 \text{ in } e_2) = 0
- \text{adepth}(\text{let } X.e) = 1 + \text{adepth}(e)

\text{anArgs}(\alpha, \beta, \gamma)\rightarrow\text{anArgs}(\alpha \rightarrow \beta, \gamma) \rightarrow (\lambda \text{ depth}(e)) \text{ operator simply counts the number of consecutive } \lambda\text{-abstractions occurring from the top of an expression } e, \text{ and } \text{anArgs}(n, \tau, \tau') \text{ adds the type } \tau \text{ to all the decorations of the functional type } \tau', \text{ up to some depth } n. \text{ The use of } \text{depth} \text{ in combination with } \text{anArgs} \text{ is important because we only want to decorate as many arrows as arguments the complete } \lambda\text{-abstraction has—corresponding to the number of arguments of the associated program rule. This avoids the incorrect decoration of arrows in the type of the right-hand side if it has a functional type, for example deriving } \text{bool} \rightarrow (\lambda \text{ depth}(e)) \text{ instead of the expected type } \text{bool} \rightarrow (\lambda \text{ depth}(e)) \text{. Notice that } \text{anArgs} \text{ is undefined for non-functional types when } n \text{ is greater than 0. However this is not a problem because in type derivations } \text{anArgs} \text{ is always applied to functional types with enough arrows (see Lemma 4 in the Appendix B of the extended version of the paper [20]) for a formal statement).}

In essence, the rule (\( \Lambda \)) derives the usual Damas-Milner types but decorating the arrows with the types of the previous arguments. For example, the expression \( \lambda X. \lambda Y. Y \)——corresponding to the \text{snd} function of Example 1——have the standard Damas-Milner type \( \alpha \rightarrow \beta \rightarrow \gamma \), while (\( \Lambda \)) derives \( \alpha \rightarrow \beta \rightarrow (\beta \rightarrow \gamma) \). By keeping the types of the arguments safely stored in the decorations, we ensure that these are always available, even for partial applications. This, together with the notion of well-typed program that we will see in the next subsection, assures that every function symbol has a transparent type.

We say that an expression \( e \) has type \( \tau \) w.r.t. \( A \) when \( A \vdash e : \tau \), and is well-typed w.r.t. \( A \)——written \( \lambda \text{in} \ A (e) \)——if \( A \vdash e : \tau \) for some type \( \tau \). Intuitively, if an expression has type \( \tau_1 \rightarrow \tau_2 \rightarrow \beta \) then \( \beta \rightarrow \tau_1 \rightarrow \tau_2 \rightarrow \beta \) corresponds to a partial application of a symbol to \( n \) expressions of types \( \tau_n \). On the other hand, types as \( \tau_1 \rightarrow \tau_2 \rightarrow \beta \) are used to allow HO parameters in functions without fixing the number of expressions (and the type) they are applied to. This situation is shown in the \text{map} function with type

\[
\forall \alpha, \beta, \gamma. (\alpha 
\rightarrow 
\beta) 
\rightarrow 
\gamma 
[\alpha] 
\rightarrow 
(\alpha 
\rightarrow 
\beta) 
[\beta]
\]

The type \( \alpha 
\rightarrow 
\beta \), of the first argument allows passing partial applications of any arity. For example, expressions as \( \text{map \ not \ true} \) and \( \text{map \ and \ true} \) [\text{false}] are well-typed, although their first argument has types \( \text{bool} \rightarrow (\text{true} \rightarrow \text{true}) \) and \( \text{bool} \rightarrow (\text{true} \rightarrow \text{false}) \) respectively.

Regarding type inference, Figure 4 shows the rules of \( \llbracket \cdot \rrbracket \). We express \( \llbracket \cdot \rrbracket \) with a relational style to show the close similarity to the type derivation \( \vdash \). However, \( A \llbracket e : \tau \rrbracket \) represents an algorithm—following the ideas of algorithm \( W \) [10]—which returns a simple type \( \tau \) and a type substitution \( \pi \) from an expression \( e \) and assumptions \( A \), failing if any of the rules cannot be applied. Intuitively, \( \tau \) is the most general type for \( e \) and \( \pi \) is the minimum substitution that \( A \) needs to be able to derive a type for \( e \). The rules of \( \llbracket \cdot \rrbracket \) are similar to those of \( \vdash \) but inserting fresh type variables in the places where type derivation guesses types, variables that can be unified during inference. Notice that, similarly to \( \vdash \), the application of \( \text{anArgs} \) in the rule (\( \Lambda \)) is always defined—see Lemma 5 in the Appendix B of the extended version of the paper [20]. The most important properties of type inference are its soundness and completeness w.r.t. type derivation:

### Theorem 1 (Soundness of \( \llbracket \cdot \rrbracket \)).
If \( A \llbracket e : \tau \rrbracket \) then \( A \pi \vdash e : \tau \).

### Theorem 2 (Completeness of \( \llbracket \cdot \rrbracket \)).
If \( A \pi \vdash e : \tau \rrbracket \) then \( A \llbracket e : \tau \rrbracket \) and there is some \( \pi' \) verifying \( A \pi' = A \pi \) and \( \pi'' \llbracket e \rrbracket = \tau \rrbracket \).
3.1 Well-typed programs

We have presented type derivation for expressions, however this notion cannot be directly extended to programs as in functional programming, because in our FLP setting let-expressions only performs pattern matching and λ-abstractions are not supported by the semantics. Therefore we need an explicit notion of well-typed program:

**Definition 2** (Well-typed program). A program rule \( f \mid \tau \rightarrow e \) is well-typed w.r.t. \( A \) if \( A \vdash \lambda X \cdot e : \tau \) and \( \tau \) is a variant of \( A(f) \). A program \( P \) is well-typed w.r.t. \( A \) —written \( wt_A(P) \) —if all its rules are well-typed w.r.t. \( A \).

The previous definition is the same as the one in [20], but using the type derivation \( \vdash \) presented in this paper. Program well-typedness proceeds rule by rule, independently of the order. Notice that forcing the derived types for the associated λ-abstraction to be a variant of the type of the function is essential to guarantee type preservation, as showed in [20].

Let us see how the proposed type system solves the opacity problems showed in Section 1. First, consider the rule for \( \text{snd} \) in Example 1. The most general type for its associated λ-abstraction \( \lambda X.\lambda Y. Y \) is \( \alpha \rightarrow (\beta) \beta \rightarrow (\alpha) \beta \) —which is 1-transparent—so the most general type that makes the rule well-typed is \( \forall \alpha, \beta. \alpha \rightarrow (\beta) \beta \rightarrow (\alpha) \beta \). This type is essentially the same as the usual type \( \forall \alpha, \beta. \alpha \rightarrow \beta \rightarrow \beta \) for \( \text{snd} \), but with the opacity removed owing to the decorations in the arrows. Therefore, the type of its application will always reveal the type of its argument: \( \lambda \text{snd} \text{true} \) can have type \( \beta \rightarrow _{(\text{true})} \beta \), \( [\text{true}] \rightarrow _{(\text{true})} [\text{true}] \), but always containing a decoration in the arrow to reveal that it is applied to a boolean. Using this type \( \forall \alpha, \beta. \alpha \rightarrow (\beta) \beta \rightarrow (\alpha) \beta \) for \( \text{snd} \), the problems with \( \text{unpack} \) (snd \( X \)) are now solved. Any valid type for \( \text{unpack} \), e.g. \( \forall \alpha, \beta. (\beta \rightarrow (\alpha) \beta) \rightarrow \alpha \), will show a connection between the type of the element contained in the pattern—which appears in the decoration of the arrow—and the result of the function. As a consequence of this connection, the \( \text{cast} \) function gets the same type as the identity function, i.e., \( \forall \alpha. \alpha \rightarrow (\alpha) \alpha \), thus recovering the synchronization between types and the behavior at the value level. This can be easily checked by looking at its associated λ-abstraction \( \lambda X.\text{unpack} \) (snd \( X \)) as, due to the types of \( \text{snd} \) and \( \text{unpack} \), the type of the right-hand side must be the same as the type of the input variable \( X \). Regarding Example 2, opaque decompositions are now solved. Any valid type for \( \text{snd} \) (\( \text{unpack} \)) will show a connection between the type of the element contained in the pattern—which appears in the decoration of the arrow—and the result of the function. As a consequence of this connection, the \( \text{cast} \) function gets the same type as the identity function, i.e., \( \forall \alpha. \alpha \rightarrow (\alpha) \alpha \), thus recovering the synchronization between types and the behavior at the value level. This can be easily checked by looking at its associated λ-abstraction \( \lambda X.\text{unpack} \) (snd \( X \)) as, due to the types of \( \text{snd} \) and \( \text{unpack} \), the type of the right-hand side must be the same as the type of the input variable \( X \).

**Theorem 3** (Type Preservation). If \( wt_A(P) \), \( A \vdash e : \tau \) and \( e \rightarrow^* e' \) then \( A \vdash e' : \tau \).

Notice that the constraint of transparent set of assumptions (which is a consequence of program well-typedness) is essential to guarantee type preservation, since it forces the type of every sub-pattern to be fixed by the type of the whole pattern. Otherwise, the steps (JoinP) or (Fapp) could easily produce ill-typed expressions, as we have seen in Examples 1 and 2.

The notion of well-typed program is based on type derivation, so it does not provide any operational mechanism to check program well-typedness or infer the types of its functions. However, we can use a type inference procedure for programs \( B \) similar to the one presented in [20]: it 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 \( P \). 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 provided by the programmer to be checked by the procedure.

**Definition 3** (Type Inference of a Program). Given a program \( \{ R_1, \ldots, R_m \} \), the procedure \( B \) for type inference of is defined as:

\[
B(A, \{ R_1, \ldots, R_m \}) = \pi, \text{if}
\]

1.  \( A \vdash (\varphi(R_1), \ldots, \varphi(R_m)) : (\tau_1, \ldots, \tau_m) \).

2.  Let \( f_1 \ldots f_n \) be the function symbols of the rules \( R_i \) in \( P \) such that \( A(f_i) \) is a closed type-scheme, and \( \tau_1^i \) the type obtained for \( R_i \) in step 1. Then \( \tau_i^* \) must be a variant of \( A(f_i) \).

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

\[
\varphi(f_1 \ldots f_n \rightarrow e) = \lambda (\lambda \ldots \lambda e) f\text{ using the special constructor pair for "tuples" of two elements of the same type, with type } \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha.
\]

The following results show that \( B \) is sound and complete w.r.t. the notion of well-typed program:

**Theorem 4** (Soundness of \( B \)). If \( B(A, \mathcal{P}) = \pi \) then \( wt_{A\pi}(\mathcal{P}) \).

**Theorem 5** (Completeness of \( B \)). If \( wt_{A\pi}(\mathcal{P}) \) then \( B(A, \mathcal{P}) = \pi \) and \( A\pi = A\pi' \) for some \( \pi' \).

Another important property of \( B \) is that, starting from transparent assumptions for constructor symbols—as it is usual for Haskell-like data declarations—the types inferred for the function symbols in the program will also be transparent, as the following transparency invariant states. This result follows easily from Theorem 4 as the substitution \( \pi \) found by \( B \) verifies \( wt_{A\pi}(\mathcal{P}) \) and every function symbol in a well-typed program w.r.t. \( A\pi \) is transparent w.r.t. \( A\pi \).

**Theorem 6** (Transparency invariant). Consider a program \( \mathcal{P} \) and a set of assumptions \( A \) satisfying the requirements of \( B \). If \( B(A, \mathcal{P}) = \pi \) then \( A\pi(f) \) is transparent for every \( f \in \mathcal{P} \).

3.2 Limitations of the type system

Although the proposed type system solves the opacity problems generated by HO patterns, as Examples 1 and 2 show, there are situations where it rejects programs that do not generate any type problems. A sample of this situation is the classical program of boolean circuits from [13], that exploits the use of HO-patterns:

**Example 3** (Boolean circuits). Consider the program

\[
x1 \ X \ Y \rightarrow X
x2 \ X \ Y \rightarrow Y
\]

\( \text{notGate} \) (\( C \) \( X \ Y \) \( \rightarrow \) \( \text{not} \) (\( C \) \( X \ Y \)) ...)
Functions $x1$ and $x2$ are basic circuits that copy one of their inputs to the output, while notGate takes a circuit as parameter and builds a circuit corresponding to the logical not gate. Using these functions to build HO patterns, it is possible to write a function that computes the size of a circuit:

\[
\begin{align*}
\text{size } x1 & \to z \\
\text{size } x2 & \to z \\
\text{size } (\text{notGate } C) & \to s (\text{size } C)
\end{align*}
\]

Using the type alias \(\text{circuit} \equiv \text{bool }\to \text{bool }\to \text{bool}\), the functions $x1$ and $x2$ have type circuit in standard Damas-Milner type systems for FLP, while notGate has type circuit \to \text{circuit}. It is clear that there is no opacity problem here, as all the types are ground. Considering these types, size would have type circuit \to \text{nat} in standard Damas-Milner. However this function is not valid in our type system since the type decorations in the types of $x1/x2$ and notGate $C$ are different. A valid type for both $x1$ and $x2$ would be

\[\text{circuit}' \equiv \text{bool }\to (\text{bool}' ) \to \text{bool}\]

Nevertheless, the type of notGate cannot be circuit' \to circuit' but a more complex type due to the decorations in the arrows:

\[
\forall \alpha, \beta. \tau \to (\tau, \alpha) \to (\tau, \beta, \text{bool}) \to \text{bool}
\]

where \(\tau \equiv \text{bool }\to \text{bool }\to \text{bool}\). With these types it is clear that \(x1/x2\) and notGate $C$ cannot have the same type, as they will differ in the type decorations:

\[
\begin{align*}
x1/x2 & : \text{bool }\to \text{bool }\to (\text{bool }\to \text{bool}) \to \text{bool} \\
\text{notGate } C & : \text{bool }\to (\text{true }, \text{bool}) \to (\text{false }, \text{true }, \text{false }, \text{true }, \text{false }, \text{false }, \text{false })
\end{align*}
\]

for some instance $\tau'$ of $\tau$. Therefore, the function size is ill-typed because there is not any type-scheme for size $\to \text{size}(\text{size } C)$. size $\to \text{size } C$ and both $\tau_1, \tau_2$ are variants of $\sigma$.

This example makes clear that if decorations in arrows reflect the arity then functions like size, whose arguments are HO patterns constructed with function symbols of different arity, will be ill-typed because their arguments will have different types. It also shows that type decorations are not always needed to remove opacity. Since some kind of type decorations are mandatory to remove opacity in some cases, it seems a good option to lighten type decorations, avoiding the use of the complete types of the previous arguments and reflecting the arity, while guaranteeing transparency in $\lambda$-abstractions. We develop this approach in the next section.

### 4. Decorations with variables

As we have seen in the previous section, if arrow decorations are a sequence of the types of previous arguments, then some functions involving HO patterns will be rejected. On the other hand, arrow decorations are essential because they provide transparency to functions, which guarantees type preservation when using the rules of evaluation (Fapp) or (JoinP). However, the transparency requirement for a function $f$ demands that $A(f)$ must be $k$-transparent if $k = \text{ar}(f) - 1$, i.e., $f(t_1 \to p_1 \dots t_k) \subseteq f(t_{\text{ftv}}(t_1 \to p_{k+1}) \to t')$ where $A(f) = \forall \sigma. \tau_1 \to p_1 \dots t_k \to p_k \to \tau_{k+1} \to p_{k+1} \to \tau'$. In other words, transparency only affects free type variables, so concrete types included in type decorations do not play any role. Therefore to avoid the undesired rejection of programs as in Example 2 and preserve types during evaluation we need to modify arrow decorations in the types of functions: instead of sequences of types we have to use sequences of type variables. These sequences of type variables must only contain the type variables occurring in the type of the previous arguments similarly as we did in the Section 3 with the whole types of the arguments. The decoration $\rho_{k+1}$ would then contain the free type variables in $\tau_1 \dots \tau_k$, so clearly the type $\forall \sigma. \tau_1 \to p_1 \dots t_k \to p_k \to \tau_{k+1} \to p_{k+1} \to \tau'$ would still be transparent.

To formalize the presented intuition we only need to modify how we derive types for $\lambda$-abstractions, forcing their arrow decorations to be sequences of type variables instead of types, as the notion of well-typed program relies on $\lambda$-abstractions to check the types of functions. Notice that this change does not modify the syntax of arrow decorations: they are still (possibly empty) sequences of simple types. We are only forcing that arrow decorations in the types of $\lambda$-abstractions—and therefore of any valid function—were sequences of type variables, but sequences of simple types can still arise as arrow decorations. For example, the expression $\text{snd } \text{true}$ must have type $\tau \to \text{bool}$ (for some $\tau$). Otherwise, it would not be possible to know from its type that it has been applied to a boolean, so an expression as $\text{snd } \text{true }\triangleright \text{snd } t$ would be incorrectly considered as well-typed. In the sequel we will use the metavariable $\chi$ to denote sequences of type variables, i.e., $\chi ::= (\overline{\chi} \tau)$ where $\overline{\chi} \equiv \chi$.

Regarding the type system, we provide a new type derivation relation $\vdash^*$ whose rules for symbols, applications and let-expressions are the same as the ones for $\vdash$ in Figure 3. For $\lambda$-abstractions, the typing rule appears in Figure 5. The $(\Lambda^\ast)$ rule is very similar to the original rule $(\Lambda)$, with the difference that it uses $\text{anArgs}^\ast$ instead of $\text{anArgs}$.

**Definition 4 (Type decoration with variables).**

\[
\text{anArgs}^\ast(n, \tau, \tau') = \tau'
\]

\[
\text{anArgs}^\ast(n, \tau_1, \tau_2) = \tau_1 \to (\chi, \chi')
\]

where $\chi'$ is the sequence of free type variables in $\tau$

The meta operator $\text{anArgs}^\ast(n, \tau, \tau')$ adds the type variables in $\tau$ to all the decorations of the functional type $\tau'$ up to some depth $n$. The sequence $\chi'$ contains the variables occurring in $\tau$ without repetitions and in the order in which they appear from left to right. To concatenate sequences of type variables we use the operator $\chi' \triangleright \chi'$ which generates a sequence without repetitions consisting of the sequence $\chi'$ followed by the sequence $\chi$ where all the variables that also appear in $\chi'$ have been removed.

In essence, the $(\Lambda^\ast)$ rule derives the usual Damas-Milner type but decorating the arrows with the type variables of the previous arguments. In some cases the type derived for a $\lambda$-abstraction using $\vdash^*$ is the same as the one derived using $\vdash$. For example, $A \vdash^* \lambda \chi. X. Y. Y' : \alpha \to (\beta \to (\chi'))$ —corresponding to the snd function in Example 2—in this case the types coincide because the sequence of type variables in $\alpha$ is the same as the whole type $\alpha$. However, if the types of the arguments are not type variables then the derived types will be different. Considering the rule for $(\Lambda)$—the boolean conjunction used in the rewriting rule (JoinP) in Figure 2—we have that $A \vdash^* \lambda \text{true}. X. Y. Y' : \text{bool }\to (\text{bool}')$. However, $A \vdash \lambda \text{true}. X. Y. Y' : \text{bool }\to (\text{bool}')$. Notice that $\vdash^*$ does not add any decoration in the second arrow, as $\text{bool}$ does not contain any type variable.

Based on this new type derivation relation, we provide a definition of well-typed programs that is the same as the one in Definition 2 but using $\vdash^*$ instead of $\vdash$. 

---

**Figure 5.** Type derivation rule for $\lambda$-abstractions.
Definition 5 (Well-typed program using \(\vdash^w\)). A program rule \(f \vdash^w e\) is well-typed w.r.t. \(A\) if \(A \vdash^w \lambda x.e : \tau\) and \(\tau\) is a variant of \(A(f)\). A program \(P\) is well-typed w.r.t. \(A\) written \(\vdash^A(P)\) if all its rules are well-typed w.r.t. \(A\).

As in the well-typed notion in the previous section, Definition 5 also assures that if \(\vdash^A(P)\) then every function in \(P\) is transparent w.r.t. \(A\). This fact is based in the following result about the types of \(\lambda\)-abstractions:

**Lemma 2.** If \(A \vdash^w \lambda x.e : \tau\) then \(\tau\) is \(n\)-transparent.

The most important property about the type system in this section is that it preserves types when evaluating expressions using well-typed programs:

**Theorem 7 (Type Preservation).** If \(\vdash^A(P)\), \(A \vdash^w e : \tau\) and \(e \rightarrow^t e'\) then \(\vdash^A(P)\).

Notice that expressions to evaluate and programs cannot contain \(\lambda\)-abstractions. As the typing rules for these expressions are the same in \(\vdash\) and \(\vdash^w\), \(A \vdash e : \tau\) is equivalent to \(A \vdash^w e : \tau\). Therefore the only difference between the previous theorem and Theorem 3 is the notion of well-typed program. However, this new notion of well-typed program is more expressive than the one in the previous section, considering as well-typed the program of boolean expressions. Notice that expressions to evaluate and programs cannot contain \(\lambda\)-abstractions.

Apart from boolean circuits, the type system in this section also covers the opacity problems presented in Section 4. Regarding the \(snd\) function in Example 4 it is still well-typed with the assumption \(\{snd : \forall a, b. \alpha \rightarrow \beta \rightarrow (a \beta)\} = A \vdash^w \alpha \rightarrow \lambda x.y.\beta\). This is transparent, solving the problem of polymorphic casting using the \texttt{unpack} function as mentioned in the previous section. Regarding the loss of type preservation using the \texttt{JoinP} and HO patterns in Example 2 the type of \(snd\) also solves the problem because of its transparency. The expression \(snd\ true \equiv snd\ z\) is still ill-typed, avoiding the problematic step \(snd\ true \equiv snd\ z \rightarrow^t true \equiv b\) that breaks type preservation. On the other hand, the rule for the \(\lambda\) function used by the let-rewriting relation (Section 2.3) is well-typed with the assumption \(\{\lambda : \alpha \rightarrow \beta\} \Rightarrow bo\ (\beta)\), \(A \vdash^w \lambda x.\texttt{true}.\lambda x.y.\beta\) should be for any \(\beta\), as \(\texttt{false}(\texttt{bo}\ (\texttt{bo}\) \(\beta))\) be computed. However, this type derivation is incorrect: the only valid type for \(snd\ true\) under \(A\) is \(\beta = \texttt{true} \Rightarrow \texttt{bo}\ (\texttt{bo}\) \(\beta\)). Therefore, the flattening process cannot take only a type to flatten but also extra information about what arrows need to be flattened, which are exactly those generated in the type of a \(\lambda\)-abstraction.

**5. Marked decorations**

To solve the lack of closure under type substitutions shown in Example 2 and being able to define a type inference algorithm both for expressions and programs, in this section we propose a new type system based on the notion of well-typed program. However, this new notion of well-typed program is more expressive than the one in the previous section, considering as well-typed the program of boolean expressions.

**Example 4.** Consider a set of assumptions \(A\) containing an assumption \(\{f : \alpha \rightarrow (\alpha)\} = A\) for the function symbol \(f\) of arity 1, and the expression \(e \equiv \lambda f.\text{true}\). We can build the type derivation

\[
\lambda e : (\alpha \rightarrow (\alpha)) \Rightarrow \text{true} \Rightarrow (\alpha) \Rightarrow \text{true} \Rightarrow (\alpha)
\]

However, using the type substitution \(\tau \equiv \alpha \Rightarrow (\alpha)\) we cannot build the type derivation \(\tau \vdash^w e : \tau\). The reason is that \(\tau \equiv (\alpha \Rightarrow (\alpha) \Rightarrow \text{true} \Rightarrow (\alpha)\)

Using the set of assumptions \(A\), the only possible type for \(e\) is \((\alpha \Rightarrow (\alpha) \Rightarrow \text{true} \Rightarrow (\alpha)\). Closure under type substitutions is an essential property of the type system because it plays an important role in the soundness of the type inference.

As in the previous section, we propose a new type system based on the notion of well-typed program. In this section, we propose a new type inference algorithm for expressions based on unification similar to \(\lll\) (thus following the same ideas as algorithm \(\mathcal{W}\) \([10]\)). As type inference for programs is based on the type system, this means that following an approach similar to \(B\) in Definition 3 would also lead to unsoundness.

Example 2 shows that the typing relation \(\vdash^w\) is not closed under type substitutions because the substitution can replace type variables in arrow decorations by types different from variables. If these variables appear in arrow decorations generated for \(\lambda\)-abstractions, the invariant that these decorations must be sequences of type variables is broken, and the type derivation \(\tau \vdash^w e : \tau\) is not possible. However, since the typing rules for symbols, applications, and let-expression are the same in \(\vdash\) and \(\vdash^w\), it is important to note that \(\vdash^w\) is closed under type substitutions for expressions no containing \(\lambda\)-abstractions.

It may seem from Example 2 that we only need to perform a “flattening” process that replaces arrow decorations by a sequence of its type variables after applying the type substitution to recover a result of closure: if \(A \vdash^w e : \tau\) then \(A \vdash^w e : \text{flat}(\tau, \pi)\). This would work in the previous example. However, the type \(\text{flat}(\tau, \pi)\) is not possible for \(\tau\) under \(\mathcal{W}\). However, not all arrow decorations should be flattened. As an example, consider a set of assumptions \(A\) containing the set of \(\{\text{snd} \equiv \forall a, b. \alpha \rightarrow \beta \rightarrow (\alpha)\beta\}\). A valid type derivation is \(\lambda x.\text{true}\) \(\Rightarrow \text{bo}\ (\text{bo})\) \(\beta\), whose type shows that this has been applied to a boolean. According to the tentative closure result proposed, the type derivation \(\tau \vdash^w \text{snd} \rightarrow^t \text{true} \Rightarrow \text{bo}\ (\text{bo})\) \(\beta\) should be for any \(\beta\), as \(\text{false}(\text{bo}\ (\text{bo}))\) \(\beta\). However, this type derivation is incorrect: the only valid type for \(\text{snd} \rightarrow^t \text{true}\) under \(\mathcal{W}\) is \(\beta = \text{bo}\ (\text{bo})\) \(\beta\). Therefore, the flattening process cannot take only a type to flatten but also extra information about what arrows need to be flattened, which are exactly those generated in the type of a \(\lambda\)-abstraction.
We use ρ∗ (marked decorations) for possibly marked arrow decorations, whereas ρ will still denote unmarked arrow decorations as in the previous sections. Similarly, we use τ∗ (marked simple types) to denote simple types with possibly marked arrow decorations and τ for simple types with unmarked arrow decorations. Marked type substitutions π∗ are finite mappings from type variables to marked simple types: π∗ := αm → τm. As types with marked arrow decorations have only been included to separate type derivation and flattening and they have not any particular meaning, in type derivations we consider that sets of assumptions contain type-schemes as defined in Figure 4, i.e., without marked arrow decorations.

Figure 6 contains the rules of the typing relation A ⊢ m e : τ∗, which is very similar to the rules in Figure 3. Notice that in (ID∗) we obtain a marked simple type τ∗ although set of assumptions contain unmarked type-schemes. The reason is that now we consider that generic instances are generated using marked type substitutions, i.e., σ ⊢ m τ∗ if σ ⊢ m τ and σ(αm → τm). The rest of rules are very similar to those in Figure 3. The main difference is (Λ∗). This rule marks the empty decoration * included in the arrow, as it is a decoration generated for a λ-abstraction. If the expression belongs to a bigger λ-abstraction this decoration will be populated with a sequence of types by the function anArgs∗, but as the whole decoration is marked with * it will be flattened. The other difference is the use of the aforementioned anArgs∗. This function is defined as anArgs in Definition 1 but considering only marked decorations:

Definition 6 (Marked type decoration).

\[ \text{anArgs}^∗(0, τ^∗_1, τ^∗_2) = τ^∗_2 \]
\[ \text{anArgs}^∗(n, τ^∗_1, τ^∗_2) = τ^∗_1 \rightarrow (τ^∗_1 \rightarrow (ϕ^∗_m \rightarrow τ^∗_2)) \quad \text{where } n > 0 \]

Notice that anArgs∗ includes simple types in arrow decorations up to some depth n, but these arrow decorations are always marked—they have been generated by the rule (Λ∗). As with anArgs, if anArgs∗(n, τ^∗_1, τ^∗_2) is used in (Λ∗) with n > 0 then τ^∗_2 is guaranteed to be a functional type with enough marked arrow decorations.

Considering the snd function, using ⊢ m we can build the following type derivation for its associated λ-abstraction: \( \lambda \cdot \lambda \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \·
we can develop a sound and complete marked type inference for expressions to be used in a type inference algorithm for programs.

Figure 7 shows the rules of the marked type inference for expressions \( \mathcal{A} \vdash_m e : \tau \). It considers marked sets of assumptions \( \mathcal{A}^* \)—i.e., set of assumptions that can contain marked simple types \( \tau^* \)—marked simple types and marked type substitutions. Like \( \vdash \), marked type inference \( \vdash_m \) has a relational style to express the similarities with \( \vdash_m \) although it is an algorithm: given a marked set of assumptions \( \mathcal{A}^* \) and expression \( e \) returns a marked simple type \( \tau^* \) and a marked type substitution \( \pi^* \), or it fails if no rule can be applied. The rules of \( \vdash_m \) are very similar to the rules of \( \vdash \) in Figure 4 with the exception of rule (i\( \Lambda^* \)). This rule follows the same ideas as the typing rule (\( \Lambda^* \)) in Figure 6 marking generated arrow decorations with \( * \) and using \( \text{anArgs} \) instead of \( \text{anArgs} \) when including types in the arrows decorations.

Intuitively, \( \vdash_m \) returns a marked simple type \( \tau^* \) that is the most general type for \( e \) and a marked type substitution \( \pi^* \) that is the minimum substitution that \( \mathcal{A}^* \) needs to be able to derive a type for \( e \) w.r.t. \( \vdash_m \). This intuition is stated in the next result:

**Theorem 10 (Properties of \( \vdash_m \) w.r.t. \( \vdash_m \))**

- **(Soundness)** If \( \mathcal{A}^* \vdash_m e : \tau^* \) then \( \mathcal{A}^* \vdash_m e : \tau \).
- **(Completeness)** If \( \mathcal{A}^* \vdash_m e : \tau^* \) then \( \mathcal{A}^* \vdash_m e : \tau^* \) and there is some \( \pi^* \) verifying \( \mathcal{A}^* \vdash \pi^* \pi^* = \mathcal{A}^* \) and \( \tau^* \pi^* = \tau^* \).

These results follow easily from soundness and completeness of \( \vdash \) (Theorems 8 and 9), as the only difference between \( \vdash_m \) and \( \vdash_m \) are the marks in arrow decorations. Notice that, although not included in the presentation of marked type derivation \( \vdash_m \), to be able to relate type inference and derivation we need to consider marked sets of assumptions for type derivations. This is not a problem, as \( \vdash_m \) supports marked set of assumptions directly (note that the only rule that uses assumptions is (ID\( ^* \)), which creates generic instances from assumptions using marked type substitutions).

Based on the type inference for expressions \( \vdash_m \), we can develop a sound—and conjectured complete—type inference algorithm for programs \( B^* \) similar to \( B \) in Section 3. As the inference algorithm \( B^* \) is sound the program is well-typed w.r.t. the resulting assumptions, so the transparency invariant holds also for \( B^* \); the types inferred for the function symbols in the program will be transparent. The definition of \( B^* \) and its properties can be found in Appendix A of the extended version of the paper [26].

### 6. Related type systems

Here we will discuss the permissiveness of our type systems compared to previous proposals of type systems for FLP; i.e., we will compare the different systems w.r.t. inclusion of the sets of well-typed programs in each typing. Regarding \([13]\), that can be considered a canonical adaptation to FLP of Damas-Milner typing, and \( \vdash \) from Section 3, none of these systems is more permissive than the other. It is shown by Example 1 (where \( \text{unpack} \) is rejected by \([13]\), as it uses an opaque pattern) and the program using boolean values will be type safe only with \( \vdash \left( \text{false} \to \text{true} \right) \) and \( \vdash \left( \text{true} \to \text{false} \right) \) is well typed with \( \vdash f : \forall \alpha. (\alpha \to \alpha) \to \text{bool} \). That program is rejected by the type systems presented in this paper because, according to the type of \( \text{snd} \left( \forall \alpha. (\alpha \to \alpha) \to \text{bool} \right) \), the types of the rules for \( f \) are not a variant of the assumption for \( f \), a condition needed to ensure permissivity. However, the type system in [20] rejects the polymorphic cast function in Example 1, accepted by all the type systems in this paper—because the rule for \( \text{unpack} \) has a critical variable. The same rules for \( f \) are accepted by [19], where permissivity can be broken freely in any pattern. However, the type system in [19] also rejects the polymorphic cast function from Example 1 because it is not type safe in absence of type decorations.

Finally, we consider that a comparison with existential types would be artificial, as our system uses HO patterns and tries to avoid opacity completely, while existential types embraces it, and usually employs first order patterns only. However it is important to remark that an approach similar to existential types \([13]\) could solve problems as the polymorphic cast function in Example 1 (it would reject the \( \text{unpack} \) function as the type systems in [19, 20]) however it cannot solve the problem of opaque decomposition (see Example 2) since the function \( \text{>\}=\text{null} \) is not defined by rules.

### 7. Conclusions and Future Work

In this paper we extend Damas-Milner typing to eliminate the unintended opacity caused by HO patterns by enhancing the expressiveness of the type language. This is different from the approach followed by previous proposals as \([13]\), where opaque patterns are forbidden from rules, or \([19, 20]\), that try to deal with opacity safely. Starting from a set of transparent assumptions for constructor symbols—as it is usual in Damas-Milner typing—the type systems presented in this paper guarantee a transparency invariant that ensures that the type derived for subsequent functions will always be transparent. As a consequence, opacity disappears from programs and type preservation is recovered, since it was destroyed just by an improper handling of opacity. Besides, by recovering transparency the problem of opaque decomposition in equalities is avoided. This is an important aspect of the paper, since (to the best of our knowledge) there is no proposal for solving the opaque decomposition problem that appear in FLP computations in the presence of HO patterns.

To eliminate the unintended opacity caused by HO patterns we enhance the expressiveness of the type language by decorating the functional type constructors \( \to \). Using these decorations we can store type information about the previous arguments of the functions so that their types are transparent. We have explored two alternatives, which differ in the amount of information stored in ar-
rows. In Section 3, we consider that arrows contains the types of all the previous arguments of the functions, i.e., they are $\forall \alpha_m. \tau_i \to \tau_j$. $\tau_2 \to \tau_3$, $\ldots$, $\tau_n \to \tau_m$. Intuitively, an expression of type $\tau_i \to \tau_j$ corresponds to a partial application of a symbol to $m$ expressions of types $\tau_m$. Although the obtained type system preserves types and enjoys a sound complete type inference algorithm for expressions, it has the drawback that it rejects some safe programs using HO patterns as Example 3. To overcome this limitation, in Section 5 we reduce the amount of type information included in arrow decorations of the types of functions to the type variables of the previous arguments. We obtain a type system that also enjoys type preservation, however it lacks an important property: closure under type substitutions. As this property is essential for developing a type inference algorithm based on unification, in Section 5 we present a type system that unites the ideas behind the two previous ones. By separating the process of deriving a marked type from the process of flattening arrow decorations we obtain a type system that enjoys type preservation, is closed under type substitution—so type inference using unification as in algorithm VW [10] is possible—and accepts the programs using HO patterns that were rejected in the first approach. All the presented type systems give a proper type to functions like the polymorphic casting function, which behaves like the identity and could only be rejected by previous proposals.

An important feature of modern FLP languages, not treated in the present work, is the support for logical variables, which are free variables that get bound during the computation by means of some narrowing mechanism [2]. The interactions between types and narrowing in FLP has not received much attention, with remarkable exceptions [3][13]. In fact the combination of logical variables and equality constraints (\(\equiv\))—which replaces the equality operator \(\!\!=\!\!\) when dealing with logical variables—allows us to reproduce all the typing problems caused by opacity of HO patterns, even in languages without them like Curry (e.g. in PAKCS 1.10.0 and MCC 0.9.11), as any program rule \(f \ p \to \ e\) with HO patterns can be emulated by a strict version \(f \ X \to \text{cond} (X \equiv \ p)\ e\). In [13] it was already detected that parametricity—there the more restrictive property of type generality is considered instead—is needed to ensure type safety with narrowing. We are convinced that our type system enjoys parametricity which, combined with the transparency warranties it provides, makes it a very promising candidate to provide a better type system for FLP with narrowing, when combined with our recent work in that subject [21]. That could improve previous proposals like [13], with tighter transparency requirements, and [3], restricted to monomorphic functions and program rules without extra variables—in contrast to [21]. We consider the approaches from [19][20] less promising for this task because of their lack of parametricity, although they could be an interesting extension when confined to some part of the program. Another possibility could be adapting existential types to FLP, but again it is not clear how might they cope with opaque decomposition.

As another line of future work we plan to integrate the type system of Section 5 into the FLP system Toy [22]. Using this system we could test the behavior of the type system with a broader set of programs. Arrow decorations are a novelty in the type language, so we wonder if they will fit easily in programmer intuition about types. Tests with the system will determine whether arrow decorations can be left as part of the types or it is better to hide them to programmers and use their information only in error messages.

---

11 Similarly to \(\equiv\), equality constraints between functional expressions are not specified in the Curry report [14] but they are supported by the mentioned Curry implementations.

8. Acknowledgments

This work has been partially supported by the Spanish projects MICINN TIN2008-06622-C03-01/TIN, CM S2009/TIC-1465 and UCM-BSCH-GR35/10-A-910502. We would also like to thank Francisco J. López-Fraguas for his help and advice with some parts of this paper, and for the interesting discussions about some topics of FLP.

References


