Transparent Function Types: Clearing up Opacity (Extended Version)*

Technical Report SIC-11-12 (Revised July 20, 2012)
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Enrique Martin-Martin
Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
emartinm@fdi.ucm.es

Juan Rodríguez-Hortalá
Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
juan.rodriguez.hortala@gmail.com

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 F.3.3 [*Logics and meanings of programs*]: Studies of Program Constructs—Type Structure; D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; D.3.1 [*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) [2, 15, 28] 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 [8, 23] or Curry [14] can be roughly described as a variant of Haskell 2010 [17] 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 [12] is the option adopted by most modern FLP implementations. Call-time choice corresponds to *call-by-need* parameter passing [4] 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 $\{coin \rightarrow z, coin \rightarrow s\ z, pair\ X \rightarrow (X, X)\}$ where z and s stand for the data constructors for Peano numbers[1]. Under a call-time choice semantics the values (z, z) and $(s\ z, s\ z)$ are correct for the expression $pair\ coin$, but the values $(z, s\ z)$ and $(s\ z, z)$ 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 [12], 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[2] logic [12]. 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 [22]. 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

* This paper is the extended version of **"Transparent Function Types: Clearing up Opacity"** that will appear in the Proceedings of the 14th International Symposium on Principles and Practice of Declarative Programming (PPDP '12).

[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 [11].

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 [3] anymore but can be distinguished by matching. For example, programmers can define different sorting functions for lists, e.g. *quicksort* and *permutsort*. They correspond to the same extensional sorting function, however, *quicksort* and *permutsort* are two different intensional descriptions that can be distinguished in the left-hand side of a rule: $\{tractable\ quicksort \to true, tractable\ permutsort \to false\}$. 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 [1, 5–7, 13, 16]. With the aim of providing standard tools for reasoning in FLP—like type-based reasoning via free theorems [29]—a new denotational semantics has been recently proposed [9]. 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 [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 [10], where the equality operator is added as a primitive of the language with type $(\bowtie) : \forall\alpha.\alpha \to \alpha \to bool$. The reason for that is twofold. First of all, overloading support by means of type classes is still in an experimental phase [24, 25]. But, more importantly, $\bowtie$ 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 $X \bowtie X$ should be reduced to $true$ as establishes the notion of joinability of HO-CRWL semantics [12]—e.g. the rule $V \bowtie V \to true$ is not valid as it is not left-linear, a usual requirement in lazy functional languages.

Nevertheless, it is well-known [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 [13, 20], illustrate these problems.

Example 1. *Consider a function snd defined by the rule*

$$snd\ X\ Y \to Y$$

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

This situation can be described using the terminology "opacity" [13], 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 when using classical Damas-Milner typing because that type system is not prepared to manage opacity safely, which can be perversely exploited to define some "polymorphic casting" function that can be used to break type safety. This can be done by using the $unpack$ function defined by the rule $unpack\ (snd\ X) \to X$, for which a direct adaptation of Damas-Milner typing gives the type $unpack : \forall\alpha,\beta.(\beta \to \beta) \to \alpha$. Note the use of the HO pattern $snd\ X$ in the definition of $unpack$. The exploit is already performed by $unpack$ and it is reflected in its type: the type α for the returning value X does not appear in the type $\beta \to \beta$ for the pattern $snd\ X$, while at the value level the returning value X is just the same X that was wrapped by snd at the input. We can use this desynchronization between types and values to define the polymorphic casting function cast *by the rule $cast\ X \to unpack\ (snd\ X)$: at the value level* cast *behaves like the identity function, as it just wraps and unwraps its input value with an application of snd, but at the type level it has the Damas-Milner type $cast : \forall\alpha,\beta.\alpha \to \beta$, because of the loss of information caused by opacity. With* cast *at hand type preservation is broken easily. For example with the usual definition for the boolean operator not, we have that $not\ (cast\ z)$ is well-typed because $cast\ z$ can be given the type* bool*, but after evaluating the call to $cast\ z$ we get the expression $not\ z$, which is ill-typed.*

Opacity is avoided in classical Damas-Milner typing by requiring constructor symbols to be transparent, i.e., not to be opaque, and in practical languages like Haskell 2010 or ML this is a consequence of the format of data type declarations. If we restrict ourselves to first order patterns this "transparency hypothesis" over constructors is enough to ensure the absence of opacity, but when HO patterns are used then opacity situations may occur, as no transparency constraint is imposed over function symbols. Another sample of the type problems generated by HO patterns can be seen in their combination with the equality operator as defined in Toy or in some Curry implementations (like PAKCS 1.10.0[3] or MCC 0.9.11[4]):[5]

Example 2. *Using the program from Example 1, the equality $snd\ z \bowtie snd\ true$ is well-typed by Damas-Milner typing because both sides admit the type $\beta \to \beta$. But to resolve that equality a decomposition step is performed, leading to the expression $z \bowtie true$ that is obviously ill-typed as it implies comparing expressions of different types. This situation was described in [13] as the problem of* opaque decomposition*, i.e., a decomposition step during the resolution of an equality in which some expressions are extracted from an opaque context. This problem is specially harmful, as the eventual occurrence of opaque decomposition is undecidable [13]. Hence, the equality operator can be used to break type preservation without the need of HO patterns in left-hand sides of rules or opaque assumptions for constructor symbols. The intuition behind this is that as the equality operator is also defined for the comparison of expressions with functional type, the definition for this primitive implicitly uses HO patterns.*

The problem in these examples is that the type system is not prepared to deal with opacity, so a solution could be extending the type system with a proper support for opacity. One option could be adapting to the FLP context the existential types extension of Damas-Milner typing [18, 26], where the transparency hypothesis is explicitly broken in data type declarations, so in this case opacity is intended in contrast with the examples above. This type system is able to handle opacity safely and it enables the creation of *abstract data types* 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,

[3] `http://www.informatik.uni-kiel.de/~pakcs/`

[4] `http://danae.uni-muenster.de/~lux/curry/`

[5] Admittedly, equality between higher-order expressions is not specified in the Curry Report [14], however, it is supported in the mentioned implementations.

the dangerous expression $snd\ z \bowtie snd\ true$ would still be a valid expression using an adaptation of existential types.

In the context of FLP, the seminal work [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 undecidable. Some other works have been developed recently to allow safe uses of opaque HO patterns [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 $n-1$ possible partial applications that conceptually correspond to $n-1$ constructor symbols whose opacity does not obey the transparency hypothesis. How do we fix this? We take inspiration from what we would do in Haskell 2010[6] for the declaration `data t B = sndc A B`, that is rejected because the variable `A` does not appear as an argument of the type constructor `t`, which implies the opaque assumption $sndc : \forall\alpha,\beta.\alpha \to \beta \to t\ \beta$—which is very similar to the type for snd in Example 1. We can easily fix this by adding `A` as an additional argument for `t`, getting the transparent assumption $sndc : \forall\alpha,\beta.\alpha \to \beta \to t\ \alpha\ \beta$. As the type constructor for functions is the arrow $\to$, we can reproduce this solution for partial applications of functions by adding a new argument to $\to$, in particular one carrying the types of the arguments already applied to the function. As we will see in this paper $\forall\alpha,\beta.\alpha \to_{()} \beta \to_{(\alpha)} \beta$ is a valid type for the function *snd* in Example 1 according to our type systems, thus clearing up the opacity of *snd*. As a consequence, the function *cast* takes the type $cast : \forall\alpha.\alpha \to_{()} \alpha$, which corresponds to its actual behaviour, and the equality $snd\ z \bowtie snd\ true$ is rejected as $\beta \to_{(nat)} \beta$ is different to $\beta \to_{(bool)} \beta$. 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. Appendices A and B contain complementary material about type inference for programs and complete proofs of the results, respectively.

[6] For the sake of notation homogeneity we employ here a modification of Haskell's syntax that uses uppercase for variables and lowercase for constructors.

Symbol	$s ::=$	$X \mid c \mid f$
Pattern	$t ::=$	$X \mid c\ \overline{t_n}\ \ (n \leq ar(c))$
	$\mid$	$f\ \overline{t_n}\ \ (n < ar(f))$
Expression	$e ::=$	$X \mid c \mid f \mid e_1\ e_2$
	$\mid$	$let\ X = e_1\ in\ e_2$
Simple type	$\tau ::=$	$\alpha \mid\ C\ \overline{\tau_n}\ (ar(C) = n)$
		$\mid \tau_1 \to_\rho \tau_2$
Type decoration	$\rho ::=$	$\alpha \mid (\tau_1, \ldots, \tau_n)\ \ (n \geq 0)$
Type-scheme	$\sigma ::=$	$\forall\overline{\alpha_n}.\tau\ \ (n \geq 0)$

Figure 1. Syntax of expressions and types

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 $ar(h)$. We also assume a denumerable set $\mathcal{DV}$ of *data variables* X. Figure 1 shows the syntax of *patterns* $\in Pat$—our notion of *value*—and *expressions* $\in Expr$. We use the notation $\overline{o_n}$ for a sequence of n syntactic objects $o_1 \ldots o_n$, where o_i refers to the i^{th} element of the sequence. If the number of elements is not relevant, we write simply $\overline{o}$. We split the set of patterns into two: *first-order patterns* $FOPat \ni fot ::= X \mid c\ \overline{fot_n}$, where $ar(c) = n$, and *higher-order patterns* $HOPat = Pat \setminus FOPat$. We also distinguish different classes of expressions: $X\ \overline{e_n}\ (n > 0)$ is a *variable application*, $c\ \overline{e_n}$ is a *junk expression* when $n > ar(c)$ and $f\ \overline{e_n}$ is an *active expression* when $n > ar(f)$. The set of *variables*—$var(e)$—and *free variables*—$fv(e)$—are defined in the usual way. Note that our let-expressions are not recursive, so $fv(let\ X = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus \{X\})$.

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

2.2 Types

We assume a denumerable set $\mathcal{TV}$ of *type variables* α and a countable alphabet $\mathcal{TC}$ of *type constructors* C. Each type constructor C has an associated arity $ar(C)$. The syntax of *simple types* τ and *type-schemes* σ 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, $bool \rightarrow bool$ and $int \rightarrow_{int} [bool]$ are syntactically invalid types, while $bool \rightarrow_{()} bool$ and $int \rightarrow_{(int)} [bool]$ are correct ones. These *type decorations* ρ in the arrows are a crucial ingredient for removing opacity from higher-order patterns as they are used to store the types of the arguments already applied in partial applications of functions, as we will see in Sections 3–5. 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: $ftv(\tau_1 \rightarrow_\rho \tau_2) = ftv(\tau_1) \cup ftv(\tau_2) \cup ftv(\rho)$. For type-schemes, $ftv(\forall\overline{\alpha}.\tau) = ftv(\tau) \setminus \{\overline{\alpha}\}$. A type-scheme $\sigma \equiv \forall\overline{\alpha}.\tau_1 \rightarrow_{\rho_1} \ldots \tau_k \rightarrow_{\rho_k} \tau'$ is called *k-transparent* if $ftv(\tau_1 \rightarrow_{\rho_1} \ldots \tau_k) \subseteq ftv(\tau')$, and *closed* if $ftv(\sigma) = \emptyset$. Notice that k-transparency implies j-transparency for any $j < k$, and every type-scheme is trivially 0-transparent.

A *set of assumptions* $\mathcal{A}$ is a set $\{\overline{s : \sigma}\}$ relating type-schemes to symbols. If $(s : \sigma) \in \mathcal{A}$ we write $\mathcal{A}(s) = \sigma$. Data constructors and function symbols appearing in $\mathcal{A}$ must be *transparent*, i.e., $\mathcal{A}(c)$ must be k_c-transparent if $k_c = ar(c)$, and $\mathcal{A}(f)$ 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 3. The transparency requirement is common over data constructors, and forbids the use of *existential types* [18, 26], but it might seem too tight for functions, at first sight. Nevertheless, as we will see in Sections 3 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 $ftv(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. The union of sets of assumptions is denoted by $\mathcal{A} \oplus \mathcal{A}'$ with the usual meaning: it contains the assumptions in $\mathcal{A}'$ as well as those in $\mathcal{A}$ for the symbols not appearing in $\mathcal{A}'$.

A *type substitution* π is a finite mapping from type variables to simple types $[\overline{\alpha \mapsto \tau}]$, whose application is defined in the usual way. The domain $dom(\pi)$ and variable range $vran(\pi)$ are defined as for data substitutions. We use ϵ to denote the identity substitution. A simple type τ' is a *generic instance* of $\sigma \equiv \forall\overline{\alpha}.\tau$ if $\tau' = \tau[\overline{\alpha \mapsto \tau''}]$ for some $\overline{\tau''}$, and we write $\sigma \succ \tau'$. Finally, we say τ' is a *variant* of $\sigma \equiv \forall\overline{\alpha}.\tau$ (written $\sigma \succ_{var} \tau'$) if $\tau' = \tau[\overline{\alpha \mapsto \beta}]$, where $\overline{\beta}$ are fresh type variables.

2.3 Operational Semantics

The operational semantics used in this work is based on let-rewriting [22]. This semantics, which is 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 ($\bowtie$) 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[7].

(LetIn) $e_1\ e_2 \rightarrow^l let\ X = e_2\ in\ e_1\ X$, if e_2 is junk, active, variable application or a let-expression; for X fresh.

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

(Elim) $let\ X = e_1\ in\ e_2 \rightarrow^l e_2$, if $X \notin fv(e_2)$

(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^l$
$let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$, if $Y \notin fv(e_3)$

(LetAp) $(let\ X = e_1\ in\ e_2)\ e_3 \rightarrow^l let\ X = e_1\ in\ e_2\ e_3$
if $X \notin fv(e_3)$

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

(JoinS) $s \bowtie s \rightarrow^l true$, if $s \in Pat$

(JoinP) $(h\ \overline{t_n}) \bowtie (h\ \overline{t'_n}) \rightarrow^l (t_1 \bowtie t'_1) \wedge \ldots \wedge (t_n \bowtie t'_n)$
if $(h\ \overline{t_n}), (h\ \overline{t'_n}) \in Pat$ and $n > 0$

(Contx) $\mathcal{C}[e] \rightarrow^l \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \rightarrow^l e'$ using any of the previous rules, and if the step is $X \bowtie X \rightarrow^l true$ using (JoinS) then $X \notin bv(\mathcal{C})$

Figure 2. Higher order let-rewriting relation with equality $\rightarrow^l$

We assume that *true* and *false* are 0-arity constructor symbols, and $\wedge$ is a binary function symbol defined with the rule $true \wedge Y \rightarrow Y$. Regarding types, we assume that every set of assumptions contains: $\{true : bool,\ false : bool,\ (\wedge) : bool \rightarrow_{()} bool \rightarrow_{(bool)} bool,\ (\bowtie) : \forall\alpha.\alpha \rightarrow_{()} \alpha \rightarrow_{(\alpha)} bool\}$. 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—$coin \bowtie coin$—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 1. Note that the rules (JoinS) and (JoinP) represent the same behavior as joinability. Therefore an equality involving two patterns $t_1 \bowtie t_2$ 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 $\bowtie$, 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 a 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.

[7] However, some of the most important Curry implementations like PAKCS or MCC currently support equality between functional expressions, following the same joinability approach.

$$\text{(ID)}\ \frac{}{\mathcal{A} \vdash s : \tau} \text{ if } \mathcal{A}(s) \succ \tau$$

$$\text{(APP)}\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to_\rho \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

$$\text{(}\Lambda\text{)}\ \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to_{()} anArgs(k, \tau_t, \tau)}$$

where $\{\overline{X_n}\} = var(t)$ and $\lambda depth(e) = k$

$$\text{(LET)}\ \frac{\mathcal{A} \vdash e_1 : \tau_x \quad \mathcal{A} \oplus \{X : \tau_x\} \vdash e_2 : \tau}{\mathcal{A} \vdash let\ X = e_1\ in\ e_2 : \tau}$$

Figure 3. Type derivation rules

We consider the type derivation relation $\vdash$ in Figure 3, where $\mathcal{A} \vdash e : \tau$ means that the expression e has type τ under the assumptions $\mathcal{A}$. 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 λ-abstractions[8] (expressions with the shape $\lambda t.e$). This kind of expressions have been included only for typing purposes, as they simplify the definition of *well-typed program rule*: a rule $f\ \overline{p_n} \to e$ is well-typed whenever its associated λ-abstraction $\lambda\overline{p_n}.e$ matches the type for f (see Definition 2). The type derivation relation in Figure 3 is based in 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 λ-abstractions, as the rule (Λ) now decorates each of the arrows in the type obtained for a chain of λ-abstractions, with the complete types of its previous arguments. For that, (Λ) uses the $\lambda depth$ and $anArgs$ meta operators:

Definition 1 (λ-depth and type decoration)**.**

$$\lambda depth(s) = 0 \qquad \lambda depth(let\ X = e_1\ in\ e_2) = 0$$
$$\lambda depth(e_1\ e_2) = 0 \quad \lambda depth(\lambda t.e) = 1 + \lambda depth(e)$$

$$anArgs(0, \tau, \tau') = \tau'$$
$$anArgs(n, \tau, \tau_1 \to_{(\overline{\tau'})} \tau_2) = \tau_1 \to_{(\tau, \overline{\tau'})} anArgs(n-1, \tau, \tau_2)$$
$$where\ n > 0$$

The $\lambda depth(e)$ operator simply counts the number of consecutive λ-abstractions occurring from the top of an expression e, and $anArgs(n, \tau, \tau')$ adds the type τ to all the decorations of the functional type τ', up to some depth n. The use of $\lambda depth$ in combination with $anArgs$ is important because we only want to decorate as many arrows as arguments the complete λ-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 $bool \to_{()} (\alpha \to_{(bool)} \beta \to_{(bool,\alpha)} \beta)$ for $\lambda true.snd$ instead of the expected type $bool \to_{()} (\alpha \to_{()} \beta \to_{(\alpha)} \beta)$. Notice that $anArgs$ is undefined for non-functional types when n is greater than 0. However this is not a problem because in type derivations $anArgs$ is always applied to functional types with enough arrows (see Lemma 4 in Appendix B for a formal statement).

[8] λ-abstractions have been excluded because there is no consensus about the semantic and operational meaning of λ-abstractions, and they are supported neither in the original let-rewriting [22] nor its underlying HO-CRWL semantics [12].

$$\text{(iID)}\ \frac{}{\mathcal{A} \Vdash s : \tau|\epsilon} \text{ if } \mathcal{A}(s) \succ_{var} \tau$$

$$\text{(iAPP)}\ \frac{\mathcal{A} \Vdash e_1 : \tau_1|\pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2|\pi_2}{\mathcal{A} \Vdash e_1\ e_2 : \alpha\pi|\pi_1\pi_2\pi}$$

where α, β fresh and $\pi = mgu(\tau_1\pi_2, \tau_2 \to_\beta \alpha)$

$$\text{(i}\Lambda\text{)}\ \frac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \Vdash e : \tau|\pi}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi \to_{()} anArgs(k, \tau_t\pi, \tau)|\pi_t\pi}$$

where $\{\overline{X_n}\} = var(t)$, $\overline{\alpha_n}$ fresh and $\lambda depth(e) = k$

$$\text{(iLET)}\ \frac{\mathcal{A} \Vdash e_1 : \tau_x|\pi_x \quad \mathcal{A}\pi_x \oplus \{X : \tau_x\} \Vdash e_2 : \tau|\pi}{\mathcal{A} \Vdash let\ X = e_1\ in\ e_2 : \tau|\pi_x\pi}$$

Figure 4. Type inference rules

In essence, the rule (Λ) derives the usual Damas-Milner type but decorating the arrows with the types of the previous arguments. For example, the expression $\lambda X.\lambda Y.Y$—corresponding to the *snd* function of Example 1—have the standard Damas-Milner type $\alpha \to \beta \to \beta$, while (Λ) derives $\alpha \to_{()} \beta \to_{(\alpha)} \beta$. 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 τ w.r.t. $\mathcal{A}$ when $\mathcal{A} \vdash e : \tau$, and is *well-typed* w.r.t. $\mathcal{A}$—written $wt_{\mathcal{A}}(e)$—if $\mathcal{A} \vdash e : \tau$ for some type τ. Intuitively, if an expression has type $\tau_1 \to_{(\overline{\tau'_m})} \tau_2$ then it corresponds to a partial application of a symbol to m expressions of types $\overline{\tau'_m}$. On the other hand, types as $\tau_1 \to_\alpha \tau_2$ 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 *map* function with type

$$\forall\alpha,\beta,\gamma.(\alpha \to_\gamma \beta) \to_{()} [\alpha] \to_{(\alpha \to_\gamma \beta)} [\beta]$$

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

Regarding type inference, Figure 4 shows the rules of $\Vdash$. We express $\Vdash$ with a relational style to show the close similarity to the type derivation $\vdash$. However $\mathcal{A} \Vdash e : \tau|\pi$ represents an algorithm—following the ideas of algorithm $\mathcal{W}$ [10]—which returns a simple type τ and a type substitution π from an expression e and assumptions $\mathcal{A}$, failing if any of the rules cannot be applied. Intuitively, τ is the most general type for e and π is the minimum substitution that $\mathcal{A}$ needs to be able to derive a type for e. The rules of $\Vdash$ 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 $anArgs$ in the rule (iΛ) is always defined—see Lemma 5 in Appendix B. The most important properties of type inference are its soundness and completeness w.r.t. type derivation:

Theorem 1 (Soundness of $\Vdash$)**.** *If* $\mathcal{A} \Vdash e : \tau|\pi$ *then* $\mathcal{A}\pi \vdash e : \tau$.

Theorem 2 (Completeness of $\Vdash$)**.** *If $\mathcal{A}\pi' \vdash e : \tau'$ then $\mathcal{A} \Vdash e : \tau|\pi$ and there is some π'' verifying $\mathcal{A}\pi\pi'' = \mathcal{A}\pi'$ and $\tau\pi'' = \tau'$.*

A trivial consequence of Theorems 1 and 2 is that when $\mathcal{A}$ is closed—$ftv(\mathcal{A}) = \emptyset$—and $\mathcal{A} \vdash e : \tau'$ then $\mathcal{A} \Vdash e : \tau|\pi$ succeeds and τ is a *principal type* of e.

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 $f\ \overline{t_n} \to e$ is well-typed w.r.t. $\mathcal{A}$ if $\mathcal{A} \vdash \lambda\overline{t_n}.e : \tau$ and τ is a variant of $\mathcal{A}(f)$. A program $\mathcal{P}$ is well-typed w.r.t. $\mathcal{A}$—written $wt_{\mathcal{A}}(\mathcal{P})$—if all its rules are well-typed w.r.t. $\mathcal{A}$.*

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 *snd* in Example 1. The most general type for its associated λ-abstraction $\lambda X.\lambda Y.Y$ is $\alpha \to_{()} \beta \to_{(\alpha)} \beta$—which is 1-transparent—so the most general type that makes the rule well-typed is $\forall \alpha, \beta.\alpha \to_{()} \beta \to_{(\alpha)} \beta$. This type is essentially the same as the usual type $\forall \alpha, \beta.\alpha \to \beta \to \beta$ for *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: $snd\ true$ can have type $\beta \to_{(bool)} \beta$, $[bool] \to_{(bool)} [bool]$... but always containing a decoration in the arrow to reveal that it is applied to a boolean. Using this type $\forall \alpha, \beta.\alpha \to_{()} \beta \to_{(\alpha)} \beta$ for *snd*, the problems with $unpack\ (snd\ X) \to X$ are now solved. Any valid type for *unpack*, e.g. $\forall \alpha, \beta.(\beta \to_{(\alpha)} \beta) \to_{()} \alpha$, 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 *cast* function gets the same type as the identity function, i.e., $\forall \alpha.\alpha \to_{()} \alpha$, 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 $\lambda X.unpack\ (snd\ X)$ as, due to the types of *snd* and *unpack*, the type of the right-hand side must be the same as the type of the input variable X. Regarding Example 2, opaque decomposition is avoided because now there is no opacity anymore. As now snd takes the transparent type $\forall \alpha, \beta.\alpha \to_{()} \beta \to_{(\alpha)} \beta$, then the expression $(snd\ true) \bowtie (snd\ z)$ is ill-typed because $snd\ true$ has type $\beta \to_{(bool)} \beta$, which is different to the type $\beta \to_{(nat)} \beta$ for $snd\ z$. Hence we will never will be forced to compare expressions with different types like $true$ and z.

Using the default set of assumptions at the beginning of Section 2.3, the rule for $\wedge$ is well-typed in every set of assumptions, since $\mathcal{A} \vdash \lambda true.\lambda Y.Y : bool \to_{()} bool \to_{(bool)} bool$, which is exactly the type assumed for $\wedge$. Similarly, the rules for *map*—$map\ F\ [\,] \to [\,]$ and $map\ F\ (X : Xs) \to (F\ X) : (map\ F\ Xs)$—are well-typed w.r.t. set of assumptions containing $map : \forall \alpha, \beta, \gamma.(\alpha \to_\gamma \beta) \to [\alpha] \to [\beta]$, the type for *map* presented before.

A fundamental remark about Definition 2 is that, if a program is well typed w.r.t. $\mathcal{A}$, then all its function symbols are transparent w.r.t. $\mathcal{A}$. This fact is based in the following result about the types of λ-abstractions (notice that 0-ary function symbols are trivially transparent):

Lemma 1. *If $\mathcal{A} \vdash \lambda\overline{t_{n+1}}.e : \tau$ then τ is n-transparent. Similarly, if $\mathcal{A} \Vdash \lambda\overline{t_{n+1}}.e : \tau'|\pi'$ then τ' is n-transparent.*

An important result of the proposed type system regarding soundness w.r.t. the semantics is *type preservation*, stating that evaluation using well-typed programs preserves types:

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

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 subpattern 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 $\mathcal{B}$ similar to the one presented in [20]: it takes a set of assumptions $\mathcal{A}$ and a program $\mathcal{P}$ and returns a type substitution π. The set $\mathcal{A}$ must contain assumptions for all the symbols in the program, even for the functions defined in $\mathcal{P}$. 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 $\mathcal{B}$ for type inference of is defined as:

$$\mathcal{B}(\mathcal{A}, \{R_1, \ldots, R_m\}) = \pi, \textit{if}$$

1. $\mathcal{A} \Vdash (\varphi(R_1), \ldots, \varphi(R_m)) : (\tau_1, \ldots, \tau_m)|\pi$.
2. *Let $f^1 \ldots f^k$ be the function symbols of the rules R_i in $\mathcal{P}$ such that $\mathcal{A}(f^i)$ is a closed type-scheme, and τ^i the type obtained for R_i in step 1. Then τ^i must be a variant of $\mathcal{A}(f^i)$.*

φ is a transformation from rules to expressions defined as:

$$\varphi(f\ t_1 \ldots t_n \to e) = pair\ (\lambda t_1. \ldots \lambda t_n.e)\ f$$

using the special constructor pair *for "tuples" of two elements of the same type, with type $\forall \alpha.\alpha \to \alpha \to \alpha$.*

The following results show that $\mathcal{B}$ is sound and complete w.r.t. the notion of well-typed program:

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

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

Another important property of $\mathcal{B}$ 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 π found by $\mathcal{B}$ verifies $wt_{\mathcal{A}\pi}(\mathcal{P})$ and every function symbol in a well-typed program w.r.t. $\mathcal{A}\pi$ is transparent w.r.t. $\mathcal{A}\pi$.

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

$$\begin{array}{l} x1\;X\;Y \to X \\ x2\;X\;Y \to Y \\ notGate\;C\;X\;Y \to not\;(C\;X\;Y) \end{array}$$

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:*

$$\begin{array}{l} size\;x1 \to z \\ size\;x2 \to z \\ size\;(notGate\;C) \to s\;(size\;C) \end{array}$$

Using the type alias circuit $\equiv$ bool $\to$ bool $\to$ bool, the functions $x1$ and $x2$ have type circuit in standard Damas-Milner type systems for FLP, while $notGate$ has type circuit $\to$ 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 $\to$ 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

$$circuit' \equiv bool \to_{()} bool \to_{(bool)} bool$$

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

$$\forall\alpha,\beta.\tau \to_{()} bool \to_{(\tau)} bool \to_{(\tau,bool)} bool$$

where $\tau \equiv bool \to_\alpha bool \to_\beta bool$. 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:

$$\begin{array}{ll} x1/x2 :: & bool \to_{()} bool \to_{(bool)} bool \\ notGate\;C :: & bool \to_{(\tau')} bool \to_{(\tau',bool)} bool \end{array}$$

for some instance τ' of τ. Therefore, the function size *is ill-typed because there is not any type-scheme for* size*—$\mathcal{A}(size) = \sigma$— such that $\mathcal{A} \vdash \lambda x1.z : \tau_1$, $\mathcal{A} \vdash \lambda(notGate\ C).s\ (size\ C) : \tau_2$ and both τ_1, τ_2 are variants of σ.*

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 λ-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 $\mathcal{A}(f)$ must be k-transparent if $k = ar(f) - 1$, i.e., $ftv(\tau_1 \to_{\rho_1} \ldots \tau_k) \subseteq ftv(\tau_{k+1} \to_{\rho_{k+1}} \tau')$ where $\mathcal{A}(f) = \forall\overline{\alpha}.\tau_1 \to_{\rho_1} \ldots \tau_k \to_{\rho_k} \tau_{k+1} \to_{\rho_{k+1}} \tau'$. In other words, transparency only affects free type variables, so concrete types included in type decorations do not play any role. Therefore

$$(\Lambda^v)\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash^v t : \tau_t \\ \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash^v e : \tau\end{array}}{\mathcal{A} \vdash^v \lambda t.e : \tau_t \to_{()} anArgs^v(k, \tau_t, \tau)}$$

$$\text{where } \{\overline{X_n}\} = var(t) \text{ and } \lambda depth(e) = k$$

Figure 5. Type derivation rule for λ-abstractions

to avoid the undesired rejection of programs as in Example 3 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 ρ_{k+1} would then contain the free type variables in $\tau_1 \ldots \tau_k$, so clearly the type $\forall\overline{\alpha}.\tau_1 \to_{\rho_1} \ldots \tau_k \to_{\rho_k} \tau_{k+1} \to_{\rho_{k+1}} \tau'$ would still be transparent.

To formalize the presented intuition we only need to modify how we derive types for λ-abstractions, forcing their arrow decorations to be sequences of type variables instead of types, as the notion of well-typed program relies on λ-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 λ-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 *snd true* must have type $\tau \to_{(bool)} \tau$ (for some τ). Otherwise, it would not be possible to know from its type that it has been applied to a boolean, so an expression as *snd true* $\bowtie$ *snd z* would be incorrectly considered as well-typed. In the sequel we will use the metavariable χ to denote sequences of type variables, i.e., $\chi ::= (\overline{\alpha_n})$ where $n \geq 0$.

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

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

$anArgs^v(0, \tau, \tau') = \tau'$
$anArgs^v(n, \tau, \tau_1 \to_{(\chi)} \tau_2) = \tau_1 \to_{(\chi' \diamond \chi)} anArgs^v(n-1, \tau, \tau_2)$
where χ' is the sequence of free type variables in τ

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

In essence, the (Λ^v) 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 λ-abstraction using $\vdash^v$ is the same as the one derived using $\vdash$. For example, $\mathcal{A} \vdash^v \lambda X.\lambda Y.Y : \alpha \to_{()} \beta \to_{(\alpha)} \beta$—corresponding to the *snd* function in Example 1. In this case the types coincide because the sequence of type variables in α is the same as the whole type α. However, if the types of the arguments are not type variables then the derived types will be different. Considering the rule for $(\wedge)$—the boolean conjunction used in the rewriting rule (JoinP) in Figure 2—we have that $\mathcal{A} \vdash^v \lambda true.\lambda Y.Y : bool \to_{()} bool \to_{()} bool$, whereas

$\mathcal{A} \vdash \lambda true.\lambda Y.Y : bool \rightarrow_{()} bool \rightarrow_{(bool)} bool$. Notice that $\vdash^v$ does not add any decoration in the second arrow, as $bool$ 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 $\vdash^v$ instead of $\vdash$.

Definition 5 (Well-typed program using $\vdash^v$).
A program rule $f\ \overline{t_n} \rightarrow e$ is well-typed w.r.t. $\mathcal{A}$ if $\mathcal{A} \vdash^v \lambda\overline{t_n}.e : \tau$ and τ is a variant of $\mathcal{A}(f)$. A program $\mathcal{P}$ is well-typed w.r.t. $\mathcal{A}$—written $wt^v_{\mathcal{A}}(\mathcal{P})$—if all its rules are well-typed w.r.t. $\mathcal{A}$.

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

Lemma 2. *If $\mathcal{A} \vdash^v \lambda\overline{t_{n+1}}.e : \tau$ then τ 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 $wt^v_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash^v e : \tau$ and $e \rightarrow^l e'$ then $\mathcal{A} \vdash^v e' : \tau$.*

Notice that expressions to evaluate and programs cannot contain λ-abstractions. As the typing rules for these expressions are the same in $\vdash$ and $\vdash^v$, $\mathcal{A} \vdash e : \tau$ is equivalent to $\mathcal{A} \vdash^v e : \tau$. 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 circuits—Example 3—as well as all the motivating programs in Section 1.

Regarding boolean circuits, consider the set of assumptions $\mathcal{A} \equiv \{x1, x2 : circuit, notGate : circuit \rightarrow_{()} circuit\}$ where $circuit \equiv bool \rightarrow_{()} bool \rightarrow_{()} bool$. With these assumptions the rules for $x1/x2$ and $notGate$ are well-typed, as $\mathcal{A} \vdash^v \lambda X.\lambda Y.X : circuit$, $\mathcal{A} \vdash^v \lambda X.\lambda Y.X : circuit$ and $\mathcal{A} \vdash^v \lambda C.\lambda X.\lambda Y.not\ (C\ X\ Y) : circuit \rightarrow_{()} circuit$. More important, these assumptions remove any information about the arity of the symbols, so the function *size* is now well-typed with the assumption $\{size : circuit \rightarrow_{()} nat\}$. The reason is that now the HO patterns $x1/x2$ and $notGate\ C$ have exactly the same type *circuit*, so all the rules for *size* have type $circuit \rightarrow_{()} nat$: $\mathcal{A} \vdash^v \lambda x1.z : circuit \rightarrow_{()} nat$, $\mathcal{A} \vdash^v \lambda x2.z : circuit \rightarrow_{()} nat$ and $\mathcal{A} \vdash^v \lambda(notGate\ C).s\ (size\ C) : circuit \rightarrow_{()} nat$.

Apart from boolean circuits, the type system in this section also solves the opacity problems presented in Section 1. Regarding the *snd* function in Example 1, it is still well-typed with the assumption $\{snd : \forall\alpha,\beta.\alpha \rightarrow_{()} \beta \rightarrow_{(\alpha)} \beta\}$ as $\mathcal{A} \vdash^v \lambda X.\lambda Y.Y : \alpha \rightarrow_{()} \beta \rightarrow_{(\alpha)} \beta$. This type is transparent, solving the problem of polymorphic casting using the *unpack* function as mentioned in the previous section. Regarding the loss of type preservation using the (JoinP) and HO patterns in Example 2, the type of *snd* also solves the problem because of its transparency. The expression $snd\ true \bowtie snd\ z$ is still ill-typed, avoiding the problematic step $snd\ true \bowtie snd\ z \rightarrow^l true \bowtie z$ that breaks type preservation. On the other hand, the rule for the $(\wedge)$ function used by the let-rewriting relation (Section 2.3) is well-typed with the assumption $\{(\wedge) : bool \rightarrow_{()} bool \rightarrow_{()} bool\}$, as $\mathcal{A} \vdash^v \lambda true.\lambda Y.Y : bool \rightarrow_{()} bool \rightarrow_{(bool)} bool$[9].

[9] As $(\wedge)$ is a predefined function used in let-rewriting rules, in this section we assume that every set of assumptions $\mathcal{A}$ contains $\{(\wedge) : bool \rightarrow_{()} bool \rightarrow_{()} bool\}$.

The type system based on $\vdash^v$ provides transparency to function symbols, solving the opacity situations that break type preservation, and it also overcomes the limitations of $\vdash$ when dealing with functions containing HO patterns as *size*. However it lacks an important property: closure under type substitutions.

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

$$\mathcal{A} \vdash^v e : (\alpha \rightarrow_{()} \alpha) \rightarrow_{()} bool \rightarrow_{(\alpha)} bool \equiv \tau$$

However, using the type substitution $\pi \equiv [\alpha \mapsto nat]$ we cannot build the type derivation $\mathcal{A}\pi \vdash^v e : \tau\pi$. The reason is that $\tau\pi \equiv (nat \rightarrow_{()} nat) \rightarrow_{()} bool \rightarrow_{(nat)} bool$ is an invalid type for a λ-abstraction using $\vdash^v$, as it decorates the final arrow with the type constructor nat instead of a sequence of type variables. Using the set of assumptions $\mathcal{A}\pi$, the only possible type for e is $(nat \rightarrow_{()} nat) \rightarrow_{()} bool \rightarrow_{()} bool$.

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. If this property does not hold, a type inference algorithm for expressions based on unification similar to $\Vdash$ (thus following the same ideas as algorithm $\mathcal{W}$ [10]) would be unsound. As type inference for programs is based on type inference for expressions, this means that following an approach similar to $\mathcal{B}$ in Definition 3 would also lead to unsoundness.

Example 4 shows that the typing relation $\vdash^v$ 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 λ-abstractions, the invariant that these decorations must be sequences of type variables is broken, and the type derivation $\mathcal{A}\pi \vdash^v e : \tau\pi$ is not possible. However, since the typing rules for symbols, applications and let-expression are the same in $\vdash$ and $\vdash^v$, it is important to note that $\vdash^v$ is closed under type substitutions for expressions no containing λ-abstractions.

It may seem from Example 4 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* $\mathcal{A} \vdash^v e : \tau$ *then* $\mathcal{A}\pi \vdash^v e : \mathit{flat}(\tau\pi)$. It would work in the previous example, since $\mathit{flat}(\tau\pi) = (nat \rightarrow_{()} nat) \rightarrow_{()} bool \rightarrow_{()} bool$, which is a valid type for e under $\mathcal{A}\pi$. However, not all arrow decorations should be flattened. As an example, consider a set of assumptions $\mathcal{A}'$ containing $\{snd :: \forall\alpha,\beta.\alpha \rightarrow_{()} \beta \rightarrow_{(\alpha)} \beta\}$. A valid type derivation is $\mathcal{A}' \vdash^v snd\ true : bool \rightarrow_{(bool)} bool$, whose type shows that it has been applied to a boolean. According to the tentative closure result proposed, the type derivation $\mathcal{A}'\pi \vdash^v snd\ true : bool \rightarrow_{()} bool$ should be correct for any π, as $\mathit{flat}((bool \rightarrow_{(bool)} bool)\pi) = bool \rightarrow_{()} bool$. However, this type derivation is incorrect: the only valid type for $snd\ true$ under $\mathcal{A}'\pi$ is $bool \rightarrow_{(bool)} bool$. 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 λ-abstraction.

5. Marked decorations

To solve the lack of closure under type substitutions shown in Example 4 and being able to define a type inference algorithm both for expressions and programs, in this section we propose a new type derivation relation $\vdash^m$. This relation is like $\vdash$ but marking with $\bullet$ those arrow decorations generated in the types of λ-abstractions. A later flattening process can then flatten marked decorations (which will be sequences of types as in $\vdash$), obtaining the same type as $\vdash^v$. The benefit of this separation of tasks is that, thanks to the

$$(\text{ID}^\bullet)\frac{}{\mathcal{A} \vdash^m s : \tau^\bullet} \text{ if } \mathcal{A}(s) \succ \tau^\bullet$$

$$(\text{APP}^\bullet)\frac{\mathcal{A} \vdash^m e_1 : \tau_1^\bullet \rightarrow_{\rho^\bullet} \tau^\bullet \quad \mathcal{A} \vdash^m e_2 : \tau_1^\bullet}{\mathcal{A} \vdash^m e_1\ e_2 : \tau^\bullet}$$

$$(\Lambda^\bullet)\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n^\bullet}\} \vdash^m t : \tau_t^\bullet \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n^\bullet}\} \vdash^m e : \tau^\bullet}{\mathcal{A} \vdash^m \lambda t.e : \tau_t^\bullet \rightarrow_{\bullet()} \mathit{anArgs}^\bullet(k, \tau_t^\bullet, \tau^\bullet)}$$

where $\{\overline{X_n}\} = \mathit{var}(t)$ and $\lambda\mathit{depth}(e) = k$

$$(\text{LET}^\bullet)\frac{\mathcal{A} \vdash^m e_1 : \tau_x^\bullet \quad \mathcal{A} \oplus \{X : \tau_x^\bullet\} \vdash^m e_2 : \tau^\bullet}{\mathcal{A} \vdash^m \mathit{let}\ X = e_1\ \mathit{in}\ e_2 : \tau^\bullet}$$

Figure 6. Marked type derivation

closure under type substitutions of $\vdash^m$, we can develop a sound and complete marking type inference algorithm $\Vdash^m$ similar to $\Vdash$. This algorithm $\Vdash^m$ could then be used in a process $\mathcal{B}^\bullet$ for inferring types of programs.

In order to handle marked arrow decorations, we consider the following new syntactic categories:

$$\begin{array}{lll} \tau^\bullet & ::= & \alpha \mid C\ \overline{\tau_n^\bullet} \text{ if } ar(C) = n \mid \tau_1^\bullet \rightarrow_{\rho^\bullet} \tau_2^\bullet \\ \rho^\bullet & ::= & \alpha \mid (\overline{\tau_n^\bullet}) \text{ if } n \geq 0 \mid {}^\bullet(\overline{\tau_n^\bullet}) \text{ if } n \geq 0 \end{array}$$

We use $\rho^\bullet$ (*marked decorations*) for possibly marked arrow decorations, whereas ρ will still denote unmarked arrow decorations as in the previous sections. Similarly, we use $\tau^\bullet$ (*marked simple types*) to denote simple types with possibly marked arrow decorations and τ for simple types with unmarked arrow decorations. *Marked type substitutions* $\pi^\bullet$ are finite mappings from type variables to marked simple types: $\pi^\bullet ::= [\overline{\alpha_n \mapsto \tau_n^\bullet}]$. 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 1, i.e., without marked arrow decorations.

Figure 6 contains the rules of the typing relation $\mathcal{A} \vdash^m e : \tau^\bullet$, which is very similar to the rules for $\vdash$ in Figure 3. Notice that in (ID$^\bullet$) we obtain a marked simple type $\tau^\bullet$ 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., $\sigma \succ \tau^\bullet$ iff $\sigma \equiv \forall\alpha_n.\tau'$ and $\tau'[\overline{\alpha_n \mapsto \tau_n^\bullet}]$. The rest of rules are very similar to those in Figure 3. The main difference is $(\Lambda^\bullet)$. This rule marks the empty decoration ${}^\bullet()$ 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 $\mathit{anArgs}^\bullet$, but as the whole decoration is marked with ${}^\bullet$ it will be flattened. The other difference is the use of the aforementioned $\mathit{anArgs}^\bullet$. This function is defined as anArgs in Definition 1 but considering only marked decorations:

Definition 6 (Marked type decoration)**.**

$\mathit{anArgs}^\bullet(0, \tau^\bullet, \tau_2^\bullet) = \tau_2^\bullet$
$\mathit{anArgs}^\bullet(n, \tau^\bullet, \tau_1^\bullet \rightarrow_{\bullet(\overline{\tau_m^\bullet})} \tau_2^\bullet) =$
$\quad \tau_1^\bullet \rightarrow_{\bullet(\tau^\bullet, \overline{\tau_m^\bullet})} \mathit{anArgs}^\bullet(n-1, \tau^\bullet, \tau_2^\bullet)$ *where* $n > 0$

Notice that $\mathit{anArgs}^\bullet$ 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 $(\Lambda^\bullet)$. As with anArgs, if $\mathit{anArgs}^\bullet(n, \tau^\bullet, \tau_2^\bullet)$ is used in $(\Lambda^\bullet)$ with $n > 0$ then $\tau_2^\bullet$ is guaranteed to be a functional type with enough marked arrow decorations.

Considering the *snd* function, using $\vdash^m$ we can build the following type derivation for its associated λ-abstraction: $\lambda X.\lambda Y.Y : \alpha \rightarrow_{\bullet()} \beta \rightarrow_{\bullet(\alpha)} \beta$. It is the same type derived using $\vdash$ but with marked arrow decorations. However a flattening process produces $\alpha \rightarrow_{()} \beta \rightarrow_{(\alpha)} \beta$, exactly the same type derived using $\vdash$ and $\vdash^v$. Regarding the operator $(\wedge)$—Section 2.3—we have the type derivation $\lambda \mathit{true}.\lambda Y.Y : \mathit{bool} \rightarrow_{\bullet()} \mathit{bool} \rightarrow_{\bullet(\mathit{bool})} \mathit{bool}$. It is the same type derived using $\vdash$ but with marked arrow decorations, and it becomes the type derived using $\vdash^v$ after flattening $(\mathit{bool} \rightarrow_{()} \mathit{bool} \rightarrow_{()} \mathit{bool})$. Considering a more complex example as function *notGate* in Example 3, a type derivation for its associated λ-abstraction is $\mathcal{A} \vdash^m \lambda C.\lambda X.\lambda Y.\mathit{not}\ C\ X\ Y : \mathit{circuit} \rightarrow_{\bullet()} \mathit{bool} \rightarrow_{\bullet(\mathit{circuit})} \mathit{bool} \rightarrow_{\bullet(\mathit{circuit},\mathit{bool})} \mathit{bool}$, where $\mathit{circuit} \equiv \mathit{bool} \rightarrow_{()} \mathit{bool} \rightarrow_{()} \mathit{bool}$—notice that *circuit* is the guessed type for C in the λ-abstraction, so it is possible to use this unmarked simple type. However, if we flatten this type we obtain the same type derived using $\vdash^v$: $\mathit{circuit} \rightarrow_{()} \mathit{circuit}$.

As showed in the previous examples, flattening a valid type obtained for e using $\vdash^m$ results in a valid type for e using $\vdash^v$. This result is clear, as $\vdash^v$ flattens decorations when using anArgs^v in the (Λ^v) rule, whereas $\vdash^m$ marks the decorations in the rule $(\Lambda^\bullet)$ and the final flattening stage obtains the same flattened arrow decorations. The following theorem formalizes this result, which uses the flattening function *flat*.

Definition 7 (Flattening)**.**

$$\begin{array}{ll} \mathit{flat}(\alpha) = & \alpha \\ \mathit{flat}(C\ \overline{\tau_n^\bullet}) = & C\ \overline{\mathit{flat}(\tau_n^\bullet)} \\ \mathit{flat}(\tau_1^\bullet \rightarrow_\rho \tau_2^\bullet) = & \mathit{flat}(\tau_1^\bullet) \rightarrow_\rho \mathit{flat}(\tau_2^\bullet) \\ \mathit{flat}(\tau_1^\bullet \rightarrow_{\bullet(\overline{\tau_m^\bullet})} \tau_2^\bullet) = & \mathit{flat}(\tau_1^\bullet) \rightarrow_\chi \mathit{flat}(\tau_2^\bullet) \end{array}$$

where χ is the sequence of type variables in $(\overline{\tau_m^\bullet})$ as they appear from left to right and without repetitions.

Theorem 8. $\mathcal{A} \vdash^m e : \tau^\bullet$ *and* $\tau = \mathit{flat}(\tau^\bullet)$ *iff* $\mathcal{A} \vdash^v e : \tau$.

As $\vdash^m$ behaves similar to $\vdash$ but marking arrow decorations that will be flattened afterward, it recovers closure under marked type substitutions:

Lemma 3 (Closure of $\vdash^m$)**.** *If* $\mathcal{A} \vdash^m e : \tau^\bullet$ *then* $\mathcal{A}\pi^\bullet \vdash^m e : \tau^\bullet\pi^\bullet$

Using the type relation $\vdash^m$ we provide a definition of well-typed programs that assures type preservation. This definition is very similar to the previous one (Definitions 2 and 5) but it needs to flatten the obtained types for λ-abstractions associated to rules:

Definition 8 (Well-typed program using $\vdash^m$)**.**
A program rule $f\ \overline{t_n} \rightarrow e$ is well-typed w.r.t. $\mathcal{A}$ if $\mathcal{A} \vdash^m \lambda\overline{t_n}.e : \tau^\bullet$, $\mathit{flat}(\tau^\bullet) = \tau$ and τ is a variant of $\mathcal{A}(f)$. A program $\mathcal{P}$ is well-typed w.r.t. $\mathcal{A}$—written $wt^m_{\mathcal{A}}(\mathcal{P})$—if all its rules are well-typed w.r.t. $\mathcal{A}$.

Theorem 9 (Type Preservation)**.** *If* $wt^m_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash^m e : \tau$ *and* $e \rightarrow^l e'$ *then* $\mathcal{A} \vdash^m e' : \tau$.

It is important to note that the new definition of well-typed program using $\vdash^m$ and flattening is equivalent to the previous one using only $\vdash^v$ (Definition 5). It follows as a simple corollary from Theorem 8. Note also that in Theorem 9 we consider only unmarked simple types for the expression e to evaluate. As expressions to evaluate cannot contain λ-abstractions, it is easy to see that valid unmarked types for these expressions using $\vdash^m$ are the same

$$(\text{iID}^\bullet)\frac{}{\mathcal{A}^\bullet \Vvdash^m s : \tau^\bullet|\epsilon} \text{ if } \mathcal{A}^\bullet(s) \succ_{var} \tau^\bullet$$

$$(\text{iAPP}^\bullet)\frac{\begin{array}{c}\mathcal{A}^\bullet \Vvdash^m e_1 : \tau_1^\bullet|\pi_1^\bullet \\ \mathcal{A}^\bullet\pi_1^\bullet \Vvdash^m e_2 : \tau_2^\bullet|\pi_2^\bullet\end{array}}{\mathcal{A}^\bullet \Vvdash^m e_1\ e_2 : \alpha\pi^\bullet|\pi_1^\bullet\pi_2^\bullet\pi^\bullet}$$

where α, β fresh and $\pi^\bullet = mgu(\tau_1^\bullet\pi_2^\bullet, \tau_2^\bullet \rightarrow_\beta \alpha)$

$$(\text{i}\Lambda^\bullet)\frac{\begin{array}{c}\mathcal{A}^\bullet \oplus \{\overline{X_n : \alpha_n}\} \Vvdash^m t : \tau_t^\bullet|\pi_t^\bullet \\ (\mathcal{A}^\bullet \oplus \{\overline{X_n : \alpha_n}\})\pi_t^\bullet \Vvdash^m e : \tau^\bullet|\pi^\bullet\end{array}}{\mathcal{A}^\bullet \Vvdash^m \lambda t.e : \tau_t^\bullet\pi^\bullet \rightarrow_{()} anArgs^\bullet(k, \tau_t^\bullet\pi^\bullet, \tau^\bullet)|\pi_t^\bullet\pi^\bullet}$$

where $\{\overline{X_n}\} = var(t)$, $\overline{\alpha_n}$ fresh and $\lambda depth(e) = k$

$$(\text{iLET}^\bullet)\frac{\begin{array}{c}\mathcal{A}^\bullet \Vvdash^m e_1 : \tau_x^\bullet|\pi_x^\bullet \\ \mathcal{A}^\bullet\pi_x^\bullet \oplus \{X : \tau_x^\bullet\} \Vvdash^m e_2 : \tau^\bullet|\pi^\bullet\end{array}}{\mathcal{A}^\bullet \Vvdash^m let\ X = e_1\ in\ e_2 : \tau^\bullet|\pi_x^\bullet\pi^\bullet}$$

Figure 7. Marked type inference rules

as valid types using $\vdash^v$, i.e., $\mathcal{A} \vdash^m e : \tau$ iff $\mathcal{A} \vdash^v e : \tau$.[10] Finally, it is important to note that $wt^m_{\mathcal{A}}(\mathcal{P})$ assures that every function in $\mathcal{P}$ is transparent w.r.t. $\mathcal{A}$, as $wt^m_{\mathcal{A}}(\mathcal{P})$ and $wt^v_{\mathcal{A}}(\mathcal{P})$ are equivalent.

5.1 Marked type inference

Up to this point, the type relation $\vdash^m$ does not provide any clear advantage over $\vdash^v$ but closure under type substitutions. However, 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 $\mathcal{A}^\bullet \Vvdash^m e : \tau^\bullet|\pi^\bullet$. It considers marked sets of assumptions $\mathcal{A}^\bullet$—i.e., set of assumptions that can contain marked simple types $\tau^\bullet$—marked simple types and marked type substitutions. Like $\Vvdash$, marked type inference $\Vvdash^m$ has a relational style to express the similarities with $\vdash^m$ although it is an algorithm: given a marked set of assumptions $\mathcal{A}^\bullet$ and expression e returns a marked simple type $\tau^\bullet$ and a marked type substitution $\pi^\bullet$, or it fails if no rule can be applied. The rules of $\Vvdash^m$ are very similar to the rules of $\Vvdash$ in Figure 4, with the exception of rule (i$\Lambda^\bullet$). This rule follows the same ideas as the typing rule ($\Lambda^\bullet$) in Figure 6, marking generated arrow decorations with $\bullet$ and using $anArgs^\bullet$ instead of $anArgs$ when including types in the arrows decorations.

Intuitively, $\Vvdash^m$ returns a marked simple type $\tau^\bullet$ that is the most general type for e and a marked type substitution $\pi^\bullet$ that is the minimum substitution that $\mathcal{A}^\bullet$ 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 $\Vvdash^m$ w.r.t. $\vdash^m$)**.**
- (Soundness) *If* $\mathcal{A}^\bullet \Vvdash^m e : \tau^\bullet|\pi^\bullet$ *then* $\mathcal{A}^\bullet\pi^\bullet \vdash^m e : \tau^\bullet$.
- (Completeness) *If* $\mathcal{A}\pi_1^\bullet \vdash^m e : \tau_1^\bullet$ *then* $\mathcal{A}^\bullet \Vvdash^m e : \tau^\bullet|\pi^\bullet$ *and there is some* $\pi_2^\bullet$ *verifying* $\mathcal{A}^\bullet\pi^\bullet\pi_2^\bullet = \mathcal{A}_1^\bullet$ *and* $\tau^\bullet\pi_2^\bullet = \tau_1^\bullet$.

These results follow easily from soundness and completeness of $\Vvdash$ (Theorem 1 and 2), as the only difference between $\vdash$/$\vdash^m$ and $\Vvdash$/$\Vvdash^m$ 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 set 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$^\bullet$), which creates generic instances from assumptions using marked type substitutions).

Based on the type inference for expressions $\Vvdash^m$, we can develop a sound—and conjectured complete—type inference algorithm for programs $\mathcal{B}^\bullet$ similar to $\mathcal{B}$ in Section 3. As the inference algorithm $\mathcal{B}^\bullet$ is sound the program is well-typed w.r.t. the resulting assumptions, so the transparency invariant holds also for $\mathcal{B}^\bullet$: the types inferred for the function symbols in the program will be transparent. The definition of $\mathcal{B}^\bullet$ and its properties can be found in Appendix A.

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 $unpack$ is rejected by [13], as it uses an opaque pattern) and the program using boolean circuits in Example 3 (accepted by [13] but not by $\vdash$). However the type system $\vdash^v$ from Section 4—which is equivalent to the type system in Section 5 regarding well-typed programs—is more permissive than the type system in [13], as it considers as well-typed the mentioned programs.

The systems from [20] and [19] also start from Damas-Milner typing and extend the system to safely support opacity. In [20] *parametricity* [27] can be broken in opaque program patterns, so a function f defined by the rules $f\ (snd\ z) \rightarrow true$ and $f\ (snd\ true) \rightarrow false\}$ is well typed with $f : \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow bool$. That program is rejected by all the type systems presented in this paper because, according to the type of *snd* ($\forall\alpha, \beta.\alpha \rightarrow_{()} \beta \rightarrow_{(\alpha)} \beta$), the types of the rules for f are not a variant of the assumption for f, a condition needed to ensure parametricity. 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 *unpack* has a *critical variable*. The same rules for f are accepted by [19], where parametricity 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 [18, 26] could solve problems as the polymorphic cast function in Example 1 (it would reject the *unpack* function as the type systems in [19, 20]) however it cannot solve the problem of opaque decomposition (see Example 2) since the function ($\bowtie$) 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

[10] As showed in Section 4, the same equivalence holds for $\vdash$ and $\vdash^v$.

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 ($\rightarrow$). 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 arrows. In Section 3 we consider that arrows contains the types of all the previous arguments of the functions, i.e., they are $\forall\overline{\alpha_m}.\tau_1 \rightarrow_{()} \tau_2 \rightarrow_{(\tau_1)} \ldots \rightarrow_{(\overline{\tau_{n-2}})} \tau_n \rightarrow_{(\overline{\tau_{n-1}})} \tau$. Intuitively, an expression of type $\tau_1 \rightarrow_{(\overline{\tau'_m})} \tau_2$ corresponds to a partial application of a symbol to m expressions of types $\overline{\tau'_m}$. Although the obtained type system preserves types and enjoys a sound a complete type inference algorithm for expressions, it has the drawback that it rejects some safe programs using HO pattens as Example 3. To overcome this limitation, in Section 4 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 $\mathcal{W}$ [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 (=:=)—which replaces the equality operator ($\bowtie$) 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 $f\ \overline{p} \rightarrow e$ with HO patterns can be emulated by a strict version[11] $f\ \overline{X} \rightarrow cond\ (\overline{X} =:= \overline{p})\ e$. 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 parts 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.

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

As another line of future work we plan to integrate the type system of Section 5 into the FLP system Toy [23]. 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.

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 want 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

[1] J. M. Almendros-Jiménez, R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. XQuery in the functional-logic language Toy. In *20th International Workshop on Functional and Constraint Logic Programming (WFLP '11)*, volume 6816 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2011.

[2] S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

[3] S. Antoy and A. Tolmach. Typed higher-order narrowing without higher-order strategies. In *Proceedings of the 4th International Symposium on Functional and Logic Programming (FLOPS '99)*, volume 1722 of *Lecture Notes in Computer Science*, pages 335–352. Springer, 1999.

[4] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '95)*, pages 233–246. ACM, 1995.

[5] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Integrating XPath with the functional-logic language toy. In *13th International Symposium on Practical Aspects of Declarative Languages (PADL '11)*, volume 6539 of *Lecture Notes in Computer Science*, pages 145–159. Springer, 2011.

[6] R. Caballero and F. López-Fraguas. A functional-logic perspective on parsing. In *Proceedings of the 4th International Symposium on Functional and Logic Programming (FLOPS '99)*, pages 85–99. Springer, 1999.

[7] R. Caballero and F. J. López-Fraguas. Functional logic parsers in Toy. Technical Report SIP-7498, Universidad Complutense de Madrid, April 1998. Available at `http://gpd.sip.ucm.es/fraguas/papers/TR-SIP-74-98-Toy-parsers.pdf`.

[8] R. Caballero, J. Sánchez, P. A. Sánchez, A. J. F. Leiva, A. G. Luezas, F. L. Fraguas, M. R. Artalejo, and F. S. Pérez. Toy, a multiparadigm declarative language. version 2.3.2, October 2011. Available at `http://toy.sourceforge.net`.

[9] J. Christiansen, D. Seidel, and J. Voigtländer. An adequate, denotational, functional-style semantics for typed FlatCurry. In *19th International Workshop on Functional and (Constraint) Logic Programming (WFLP '10), Revised Selected Papers*, volume 6559 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2011.

[10] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*, pages 207–212. ACM, 1982.

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

[12] J. González-Moreno, T. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proceedings of the 14th International Conference on Logic Programming (ICLP '97)*, pages 153–167. MIT Press, 1997.

[13] J. González-Moreno, T. Hortalá-González, and M. Rodríguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1), July 2001.

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

[15] M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP '07)*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer, 2007.

[16] T. Hortalá-González, F. J. López-Fraguas, J. Sánchez-Hernández, and E. Ullán-Hernández. Declarative programming with real constraints. Technical Report SIP-5997, Universidad Complutense de Madrid, April 1997. Available at `http://gpd.sip.ucm.es/fraguas/papers/TR-DIA-5997-CFLPR.pdf`.

[17] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*, pages 12–1–12–55. ACM, 2007.

[18] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16:1411–1430, 1994.

[19] F. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. Liberal typing for functional logic programs. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS '10)*, volume 6461 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2010.

[20] F. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. New results on type systems for functional logic programming. In *18th International Workshop on Functional and (Constraint) Logic Programming (WFLP '09), Revised Selected Papers*, volume 5979 of *Lecture Notes in Computer Science*, pages 128–144. Springer, 2010.

[21] F. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. Well-typed narrowing with extra variables in functional-logic programming. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '12)*, pages 83–92. ACM, 2012.

[22] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS '08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2008.

[23] F. López-Fraguas and J. Sánchez-Hernández. Toy: A multiparadigm declarative system. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA '99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 244–247. Springer, 1999.

[24] W. Lux. Adding Haskell-style overloading to Curry. In *Workshop of Working Group 2.1.4 of the German Computing Science Association GI*, pages 67–76, 2008.

[25] E. Martin-Martin. Type classes in functional logic programming. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '11)*, pages 121–130. ACM, 2011.

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

[27] J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, (83):513–523, 1983.

[28] M. Rodríguez-Artalejo. Functional and constraint logic programming. In *Constraints in Computational Logics: Theory and Applications. International Summer School (CCL99)*, volume 2002 of *Lecture Notes in Computer Science*, pages 202–270. Springer, 2001.

[29] P. Wadler. Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, pages 347–359. ACM, 1989.

A. Type inference for programs using $\Vdash^m$

In this section we will present a type inference algorithm for programs $\mathcal{B}^\bullet$ based on marked type inference for expressions $\Vdash^m$. This algorithm, which follows the same ideas as $\mathcal{B}$ in Definition 3, is sound w.r.t. the notion of well-typed program: the resulting type substitution computed by the algorithm makes the program well-typed. Regarding completeness, we do not have a formal proof yet but we conjecture that whenever a program can be well-typed applying an unmarked type substitution to the set of assumptions, the algorithms succeeds and finds a marked type substitution that is "more general".

The type inference algorithm for programs is defined as follows:

Definition 9 (Type inference of a program)**.**
Consider a set of assumptions $\mathcal{A}$ containing a closed type-scheme as assumption for every symbol s that is not a defined function in $\mathcal{P}$, and $\{f : \alpha\}$ (with α fresh) for every function f defined in $\mathcal{P}$. The procedure $\mathcal{B}^\bullet$ for type inference of a program $\{R_1, \ldots, R_m\}$ is defined as:

$$\mathcal{B}^\bullet(\mathcal{A}, \{R_1, \ldots, R_m\}) = \pi^\bullet, \ \textit{if} \quad \mathcal{A} \Vdash^m (\varphi(R_1), \ldots, \varphi(R_m)) : (\tau_1^\bullet, \ldots, \tau_m^\bullet)|\pi^\bullet$$

where φ is the same transformation from rules to pairs of λ-abstractions and function symbols used in $\mathcal{B}$ (Definition 3), i.e.,

$$\varphi(f\, t_1 \ldots t_n \to e) = pair\ (\lambda t_1. \ldots \lambda t_n.e)\ f$$

and pair *is a special constructor with type $\forall\alpha.\alpha \to \alpha \to \alpha$ for "tuples" of two elements of the same type.*

The resulting type inference procedure for programs $\mathcal{B}^\bullet$ is slightly less expressive than $\mathcal{B}$ because it requires that every function in $\mathcal{P}$ has a type variable as assumption in $\mathcal{A}$. If closed type-scheme were provided for some function symbols then $\mathcal{B}^\bullet$ would fail for their rules: the inferred type for the λ-abstraction would have marks whereas the inferred type for the function symbol would never contain marks (remember that type-schemes do not contain marks), so they would not be able to unify. Therefore $\mathcal{B}^\bullet$ can only be used to infer types of function symbols, and it cannot be used to check whether the types provided by the programmer are correct or not. This situation shows that a naive adaptation of $\mathcal{B}$ using $\Vdash^m$ is not as expressive as desired, so exploring variants of this approach to recover the lost typechecking capabilities is an interesting subject of future work. Applying a checking phase after the type inference $\mathcal{B}^\bullet$ defined above to check that declared types are not more general than the inferred ones seems less promising than exploring other variants that use the declared types while inferring. The reason is that if declared types are not considered while inferring, type inference in programs with *polymorphic recursion* could fail in some situations.

Apart from the mentioned limitation the procedure $\mathcal{B}^\bullet$ is sound, i.e., the marked type substitution found makes the program well-typed:

Theorem 11 (Soundness of $\mathcal{B}^\bullet$)**.** *Consider that $\mathcal{A}$ contains only closed type-schemes and type variables as assumptions for function symbols defined in $\mathcal{P}$. If $\mathcal{B}^\bullet(\mathcal{A}, \mathcal{P}) = \pi^\bullet$ and $\pi' = flat(\pi^\bullet)$ then $wt^m_{\mathcal{A}\pi'}(\mathcal{P})$.*

As the rest of results in this paper, the complete proof can be found in Appendix B. Flattening marked type substitutions is defined as

$$flat([\overline{\alpha_n \mapsto \tau_n^\bullet}]) = [\overline{\alpha_n \mapsto flat(\tau_n^\bullet)}]$$

Notice that we flatten the type substitution $\pi^\bullet$ obtained from $\mathcal{B}^\bullet$, so the resulting set of assumptions $\mathcal{A}\pi'$ cannot contain marked assumptions, as desired.

Regarding completeness of the type inference procedure for programs $\mathcal{B}^\bullet$, we conjecture that it is complete in the same sense of the original $\mathcal{B}$:

Conjecture 1 (Completeness of $\mathcal{B}^\bullet$)**.** *Consider a set of assumptions $\mathcal{A}$ containing only closed type-schemes and type variables for function symbols defined in $\mathcal{P}$. If $\pi' = flat(\pi^\bullet)$ and $wt^m_{\mathcal{A}\pi'}(\mathcal{P})$ then $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi_g^\bullet$ such that $\mathcal{A}\pi^\bullet = \mathcal{A}\pi_g^\bullet\pi''$ for some π''.*

Notice that $\mathcal{B}^\bullet$ cannot find a more general type substitution than the one that makes the program well-typed (π') since it computes a type substitution $\pi_g^\bullet$ whose range are marked simple types (from λ-abstraction) and by definition of well-typed program the range of π' must be unmarked simple types. For example the program $\{f : true \to false\}$ is well-typed w.r.t. $\{f : \alpha\}[\alpha \mapsto bool \to_{()} bool]$ but the type substitution found by $\mathcal{B}^\bullet$ is $[\alpha \mapsto bool \to_{\bullet()} bool]$, which cannot be compared to $[\alpha \mapsto bool \to_{()} bool]$ because of the mark in the arrow decoration.

B. Proofs

B.1 Auxiliary results

Lemma 4. *If $\mathcal{A} \vdash e : \tau$ and $\lambda depth(e) = n > 0$ then*

$$\tau \equiv \tau_1' \to_{()} \tau_2' \to_{(\tau_1')} \tau_3' \to_{(\tau_1', \tau_2')} \ldots \tau_n' \to_{(\overline{\tau_{n-1}'})} \tau'$$

Proof. By induction on n. □

Lemma 5. *If $\mathcal{A} \Vdash e : \tau|\pi$ and $\lambda depth(e) = n > 0$ then*

$$\tau \equiv \tau_1' \to_{()} \tau_2' \to_{(\tau_1')} \tau_3' \to_{(\tau_1', \tau_2')} \ldots \tau_n' \to_{(\overline{\tau_{n-1}'})} \tau'$$

Proof. By induction on n. □

Lemma 6. *If $anArgs(n, \tau, \tau')$ is defined then $anArgs(n, \tau, \tau')\pi = anArgs(n, \tau\pi, \tau'\pi)$*

Proof. By induction on n. □

Notice that there are cases where $anArgs(n, \tau, \tau')\pi \not\equiv anArgs(n, \tau\pi, \tau'\pi)$. For example, taking $\pi \equiv [\alpha \mapsto int \to_{()} int]$ we have that $anArgs(1, bool\pi, \alpha\pi) \equiv anArgs(1, bool, int \to_{()} int) \equiv int \to_{(bool)} int$. However, $anArgs(n, bool, \alpha)$ is not defined and neither is $anArgs(n, bool, \alpha)\pi$.

Lemma 7 (Properties of $\vdash$)**.**

a) If $\mathcal{A} \vdash e : \tau$ then $\mathcal{A}\pi \vdash e : \tau\pi$.
b) Let s be a symbol not appearing in e, and σ_s any type-scheme. Then $\mathcal{A} \vdash e : \tau \Leftrightarrow \mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$.
c) Assume $\mathcal{A}(X) = \tau_x$ and $\mathcal{A} \vdash e' : \tau_x$. If $\mathcal{A} \vdash e : \tau$ then $\mathcal{A} \vdash e[X \mapsto e'] : \tau$.

Proof. The proofs are very similar to those in [20]:

a) By induction over the structure of e, using Lemma 6 in the (Λ) case.
b) Both directions are proved by induction over the size of the derivation tree.
c) By induction over the size of the derivation tree, using *b)*.

$\square$

B.2 Proof of Theorem 1 (Soundness of $\Vdash$)

Theorem 1 (Soundness of $\Vdash$). *If $\mathcal{A} \Vdash e : \tau|\pi$ then $\mathcal{A}\pi \vdash e : \tau$.*

Proof. By induction over the type inference, using Lemma 7-a. $\square$

B.3 Proof of Theorem 2 (Completeness of $\Vdash$)

To prove Theorem 2, we will use the following remark about type inference:

Remark 1 (Uniqueness of the type inference)**.** *The result of a type inference is unique upon renaming of fresh type variables. In a type inference $\mathcal{A} \Vdash e : \tau|\pi$ the variables in $ftv(\tau)$, $dom(\pi)$ or $vran(\pi)$ that do not occur in $ftv(\mathcal{A})$ are fresh variables generated by the inference process, so the result remains valid changing those variables by any other fresh types variables.*

Theorem 2 (Completeness of $\Vdash$). *If $\mathcal{A}\pi' \vdash e : \tau'$ then $\mathcal{A} \Vdash e : \tau|\pi$ and there is some π'' verifying $\mathcal{A}\pi\pi'' = \mathcal{A}\pi'$ and $\tau\pi'' = \tau'$.*

Proof. By induction over the type derivation. The proof is similar to the proof for completeness in [20]. Therefore we only detail the most interesting cases:

$\boxed{\text{(APP)}}$ We have the type derivation:

$$\text{(APP)}\frac{\begin{array}{c}\mathcal{A}\pi' \vdash e_1 : \tau'_1 \to_\rho \tau'_2 \ (A) \\ \mathcal{A}\pi' \vdash e_2 : \tau'_1 \ (B)\end{array}}{\mathcal{A}\pi' \vdash e_1\ e_2 : \tau'_2}$$

From (A) by the Induction Hypothesis we have $\mathcal{A} \Vdash e_1 : \tau_1|\pi_1$ and there is a type substitution π''_1 such that (C) $\tau_1\pi''_1 = \tau'_1 \to_\rho \tau'_2$ and $\mathcal{A}\pi_1\pi''_1 = \mathcal{A}\pi'$. We can write (B) as $\mathcal{A}\pi_1\pi''_1 \vdash e_2 : \tau'_1$ so again by the Induction Hypothesis $\mathcal{A}\pi_1 \Vdash e_2 : \tau_2|\pi_2$ and for some π''_2 we have (D) $\mathcal{A}\pi_1\pi_2\pi''_2 = \mathcal{A}\pi_1\pi''_1$ and $\tau_2\pi''_2 = \tau'_1$. We consider that π''_2 is minimal in the sense that $dom(\pi''_2) \subseteq ftv(\tau_2) \cup ftv(\mathcal{A}\pi_1\pi_2)$. In order to build the type derivation, we need that $\tau_1\pi_2$ and $\tau_2 \to_\beta \alpha$ unify, being α and β fresh type variables. For that we will find a type substitution π_u that unifies these two types. Consider the set of type variables $B \equiv dom(\pi''_1) \smallsetminus ftv(\mathcal{A}\pi_1)$ and $\pi_u \equiv \pi''_1|_B + \pi''_2 + [\alpha \mapsto \tau'_2, \beta \mapsto \rho]$, which is well defined since the constituent substitutions have disjoint domains. The type variables α and β are fresh, so they cannot appear in $dom(\pi''_1|_B)$ or $dom(\pi''_2)$. To prove that $dom(\pi''_1|_B)$ and $dom(\pi''_2)$ have disjoint domains, consider a type variable $\gamma \in dom(\pi''_2)$. We have two cases:

- If $\gamma \in ftv(\mathcal{A}\pi_1)$ then $\gamma \notin B$, so $\gamma \notin dom(\pi''_t|_B)$.
- If $\gamma \notin ftv(\mathcal{A}\pi_1)$ then it must occur in $vran(\pi_2)$ or in $ftv(\tau_2)$, since π''_2 is minimal. Therefore by Remark 1 γ is a fresh variable and it cannot appear in $dom(\pi''_1)$, so $\gamma \notin dom(\pi''_1|_B)$.

We now prove that π_u is an unifier of $\tau_1\pi_2$ and $\tau_2 \to_\beta \alpha$, i.e., $\tau_1\pi_2\pi_u = \tau'_2 \to_\rho \tau'_2 = (\tau_2 \to_\beta \alpha)\pi_u$. To prove the first equality consider a type variable $\gamma \in ftv(\tau_1)$:

- If $\gamma \in ftv(\mathcal{A}\pi_1)$ then consider a type variable $\delta \in ftv(\gamma\pi_2)$:
  - If $\delta \in ftv(\mathcal{A}\pi_1)$ then $\delta \notin B$, so $\delta\pi_u = \delta\pi''_2$.
  - If $\delta \notin ftv(\mathcal{A}\pi_1)$ then by Remark 1 δ is a fresh variable—notice that $\delta \in vran(\pi_e)$, otherwise $\delta \equiv \gamma$ which is a contradiction because $\gamma \in ftv(\mathcal{A}\pi_1)$—so it cannot appear in $dom(\pi''_1)$, therefore $\delta\pi_u = \delta\pi''_2$.

  We have that $\gamma\pi_2\pi_u = \gamma\pi_2\pi''_2$, and by (D) $\gamma\pi_2\pi''_2 = \gamma\pi''_1$.
- If [a] $\gamma \notin ftv(\mathcal{A}\pi_1)$ then by Remark 1 we have [b] $\gamma \notin dom(\pi_2)$, [c] $\gamma \notin vran(\pi_2)$ and [d] $\gamma \notin ftv(\tau_2)$—remember that $\gamma \in ftv(\tau_1)$. By [a] and [c] then [e] $\gamma \notin ftv(\mathcal{A}\pi_1\pi_2)$. Since π''_2 is minimal, then by [d] and [e] we have [f] $\gamma \notin dom(\pi''_2)$. Therefore $\gamma\pi_2\pi_u = \gamma\pi_u$ (by [b]) and $\gamma\pi_u = \gamma(\pi''_1|_B)$ (by [f], since α and β are fresh). Finally, $\gamma(\pi''_1|_B) = \gamma\pi''_1$ (by [a]), so $\gamma\pi_2\pi_u = \gamma\pi''_1$.

From the previous case distinction we can conclude that $\tau_1\pi_2\pi_u = \tau_1\pi_1'' = \tau_1' \to_\rho \tau_2'$ (by C). On the other hand, we have to prove $(\tau_2 \to_\beta \alpha)\pi_u = \tau_1' \to_\rho \tau_2'$. The equalities $\alpha\pi_u = \tau_2'$ and $\beta\pi_u = \rho$ follow from definition of π_u, so we only need to treat the case $\tau_2\pi_u = \tau_1'$. Consider a type variable $\gamma \in \mathit{ftv}(\tau_2)$:

- If $\gamma \in \mathit{ftv}(\mathcal{A}\pi_1)$ then $\gamma \notin B$, so $\gamma \notin \mathit{dom}(\pi_1''|B)$. Therefore $\gamma\pi_u = \gamma\pi_2''$.
- If $\gamma \notin \mathit{ftv}(\mathcal{A}\pi_1)$ then by Remark 1 it is fresh and it cannot occur in $\mathit{dom}(\pi_1'')$, so $\gamma \notin \mathit{dom}(\pi_1''|B)$. Therefore $\gamma\pi_u = \gamma\pi_2''$.

By the previous case distinction we have that $\tau_2\pi_u = \tau_2\pi_2''$, so by (D) $\tau_2\pi_2'' = \tau_1'$.

Since the substitution π_u is a unifier of $\tau_1\pi_2$ and $\tau_2 \to_\beta \alpha$ then there is a most general unifier $\pi = \mathit{mgu}(\tau_1\pi_2, \tau_2 \to_\beta \alpha)$ such that $\pi\pi'' = \pi_u$. Therefore the following type inference is correct:

$$\text{(iAPP)}\frac{\begin{array}{c}\mathcal{A} \Vdash e_1 : \tau_1|\pi_1 \\ \mathcal{A}\pi_1 \Vdash e_2 : \tau_2|\pi_2\end{array}}{\mathcal{A} \Vdash e_1\ e_2 : \alpha\pi|\pi_1\pi_2\pi}$$

Clearly the substitution π'' such that $\pi_u = \pi\pi''$ verifies $\alpha\pi\pi'' = \tau_2'$. It also verifies $\mathcal{A}\pi_1\pi_2\pi\pi'' = \mathcal{A}\pi'$. To prove it, consider a type variable $\gamma \in \mathit{ftv}(\mathcal{A}\pi_1)$:

- If $\gamma \in \mathit{dom}(\pi_2)$ then consider a type variable $\delta \in \mathit{ftv}(\gamma\pi_2)$:
  - If $\delta \in \mathit{ftv}(\mathcal{A}\pi_1)$ then $\delta \notin B$, so $\delta \notin \mathit{dom}(\pi_1''|_B)$ and $\delta\pi_u = \delta\pi_2''$.
  - If $\delta \notin \mathit{ftv}(\mathcal{A}\pi_1)$ then by Remark 1 δ is a fresh variable and cannot appear in $\mathit{dom}(\pi_1''|_B)$—notice that $\delta \in \mathit{vran}(\pi_2)$, otherwise $\gamma \equiv \delta$ which is a contradiction because $\gamma \in \mathit{ftv}(\mathcal{A}\pi_1)$—so $\delta\pi_u = \delta\pi_2''$.

  Therefore for every $\delta \in \mathit{ftv}(\gamma\pi_2)$ we have that $\delta\pi_u = \delta\pi_2''$, so $\gamma\pi_2\pi_u = \gamma\pi_2\pi_2''$.
- If $\gamma \notin \mathit{dom}(\pi_2)$ then $\gamma\pi_2\pi_u = \gamma\pi_u$. Since $\gamma \in \mathit{ftv}(\mathcal{A}\pi_1)$ then $\gamma \notin B$ so $\gamma \notin \mathit{dom}(\pi_1''|_B)$. Therefore $\gamma\pi_u = \gamma\pi_2'' = \gamma\pi_2\pi_2''$.

By the previous case distinction we have that every type variable $\gamma \in \mathit{ftv}(\mathcal{A}\pi_1)$ verifies $\gamma\pi_2\pi_u = \gamma\pi_2\pi_2''$, therefore $\mathcal{A}\pi_1\pi_2\pi\pi'' = \mathcal{A}\pi_1\pi_2\pi_u = \mathcal{A}\pi_1\pi_2\pi_2'' = \mathcal{A}\pi_1\pi_1'' = \mathcal{A}\pi'$ by (D) and (C).

$\boxed{(\Lambda)}$ We have the type derivation:

$$(\Lambda)\frac{\begin{array}{c}\mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t'\ (A) \\ \mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau_e'\ (B)\end{array}}{\mathcal{A}\pi' \vdash \lambda t.e : \tau_t' \to_{()} \mathit{anArgs}(m, \tau_t', \tau_e')}$$

where $\lambda \mathit{depth}(e) = m$. Let $\overline{\mathcal{A}_n}$ be fresh type variables, and $\pi_g \equiv [\overline{\alpha_n \mapsto \tau_n}]$. We can write (A) as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'\pi_g \vdash t : \tau_t'$, so by the Induction Hypothesis we have $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t$ and there is a substitution π_t'' such that (C) $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_t'' = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'\pi_g$—i.e., $\mathcal{A}\pi_t\pi_t'' = \mathcal{A}\pi'$ and $\alpha_i\pi_t\pi_t'' = \tau_i$—and $\tau_t\pi_t'' = \tau_t'$.

Using these equalities we can write (B) as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_t'' \vdash e : \tau_e'$, and again by the Induction Hypothesis we have $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \Vdash e : \tau_e|\pi_e$ and some π_e'' verifying (D) $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_e\pi_e'' = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_t''$—i.e., $\mathcal{A}\pi_t\pi_e\pi_e'' = \mathcal{A}\pi_t\pi_t''$ and $\alpha_i\pi_t\pi_e\pi_e'' = \alpha\pi_t\pi_t''$—and $\tau_e\pi_e'' = \tau_e'$. We consider that π_e'' is minimal in the sense that $\mathit{dom}(\pi_e'') \subseteq \mathit{ftv}(\tau_e) \cup \mathit{ftv}((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_e)$.

With the previous type inferences we can build the type inference of the λ-abstraction—notice that by Lemma 5 $\mathit{anArgs}(m, \tau_t\pi_e, \tau_e)$ is defined:

$$(\text{i}\Lambda)\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t \\ (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \Vdash e : \tau_e|\pi_e\end{array}}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi_e \to_{()} \mathit{anArgs}(m, \tau_t\pi_e, \tau_e)|\pi_t\pi_e}$$

We have to find a substitution π'' such that $\mathcal{A}\pi_t\pi_e\pi'' = \mathcal{A}\pi'$ and $(\tau_t\pi_e \to_{()} \mathit{anArgs}(m, \tau_t\pi_e, \tau_e))\pi'' = \tau_t' \to_{()} \mathit{anArgs}(m, \tau_t', \tau_e')$. Let B be the set of type variables defined as $B \equiv \mathit{dom}(\pi_t'') \setminus \mathit{ftv}((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$, and $\pi'' \equiv \pi_t''|_B + \pi_e''$. The substitution π'' is well defined because $\pi_t''|_B$ and π_e'' have disjoint domains. To prove it, consider a type variable $\alpha \in \mathit{dom}(\pi_e'')$. We have two cases:

- If $\alpha \in \mathit{ftv}((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$ then $\alpha \notin B$, so $\alpha \notin \mathit{dom}(\pi_t''|_B)$.
- If $\alpha \notin \mathit{ftv}((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$ then it must occur in $\mathit{vran}(\pi_e)$ or in $\mathit{ftv}(\tau_e)$, since π_e'' is minimal. Therefore by Remark 1 α is a fresh variable and it cannot appear in $\mathit{dom}(\pi_t'')$, so $\alpha \notin \mathit{dom}(\pi_t''|_B)$.

In order to prove that $\mathcal{A}\pi_t\pi_e\pi'' = \mathcal{A}\pi'$ we need to prove first that for every type variable $\alpha \in \mathit{ftv}(\mathcal{A}\pi_t)$ then $\alpha\pi_e\pi'' = \alpha\pi_e\pi_e''$:

- If $\alpha \in \mathit{dom}(\pi_e)$ then consider a type variable $\delta \in \mathit{ftv}(\alpha\pi_e)$:
  - If $\delta \in \mathit{ftv}((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$ then $\delta \notin B$, so $\delta\pi'' = \delta\pi_e''$.
  - If $\delta \notin \mathit{ftv}((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$ then by Remark 1 α is a fresh variable—notice that $\delta \in \mathit{vran}(\pi_e)$, otherwise $\delta \equiv \alpha$ which is a contradiction because $\alpha \in \mathit{ftv}(\mathcal{A}\pi_t)$—and it cannot appear in $\mathit{dom}(\pi_t'')$, so $\delta\pi'' = \delta\pi_e''$.

  Therefore $\alpha\pi_e\pi'' = \alpha\pi_e\pi_e''$.
- If $\alpha \notin \mathit{dom}(\pi_e)$ then $\alpha\pi_e\pi'' = \alpha\pi''$. Since $\alpha \in \mathit{ftv}(\mathcal{A}\pi_t)$ then $\alpha \notin B$, so $\alpha\pi'' = \alpha\pi_e'' = \alpha\pi_e\pi_e''$.

So we can conclude that $\mathcal{A}\pi_t\pi_e\pi'' = \mathcal{A}\pi_t\pi_e\pi''_e = \mathcal{A}\pi_t\pi''_t = \mathcal{A}\pi'$ by (D) and (C).

In order to prove that $(\tau_t\pi_e \rightarrow_{()} anArgs(m, \tau_t\pi_e, \tau_e))\pi'' = \tau'_t \rightarrow_{()} anArgs(m, \tau'_t, \tau'_e)$ we need to prove that $\tau_t\pi_e\pi'' = \tau'_t$ and $\tau_e\pi'' = \tau'_e$. To prove the first equality consider a type variable $\alpha \in ftv(\tau_t)$:

- If $\alpha \in ftv((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$ then consider a variable $\delta \in ftv(\alpha\pi_e)$:
  - If $\delta \in ftv((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$ then $\delta \notin B$, so $\delta\pi'' = \delta\pi''_e$.
  - If $\delta \notin ftv((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$ then by Remark 1 δ is a fresh variable—notice that $\delta \in vran(\pi_e)$, otherwise $\delta \equiv \alpha$ which is a contradiction because $\alpha \in ftv((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$—so it cannot appear in $dom(\pi''_t)$, therefore $\delta\pi'' = \delta\pi''_e$.

  We have $\alpha\pi_e\pi'' = \alpha\pi_e\pi''_e$, and by(D) $\alpha\pi_e\pi''_e = \alpha\pi''_t$.
- If [a] $\alpha \notin ftv((\mathcal{A}\oplus\{\overline{X_n : \alpha_n}\})\pi_t)$ then by Remark 1 we have [b] $\alpha \notin dom(\pi_e)$, [c] $\alpha \notin vran(\pi_e)$ and [d] $\alpha \notin ftv(\tau_e)$—remember that $\alpha \in ftv(\tau_t)$. By [a] and [c] then [e] $\alpha \notin ftv((\mathcal{A}\oplus\{\overline{X_n : \alpha_n}\})\pi_t\pi_e)$. Since π''_e is minimal, then by [d] and [e] we have [f] $\alpha \notin dom(\pi''_e)$. Therefore $\alpha\pi_e\pi'' = \alpha\pi''$ (by [b]) and $\alpha\pi'' = \alpha(\pi''_t|_B)$ (by [f]). Finally, $\alpha(\pi''_t|_B) = \alpha\pi''_t$ (by [a]), so $\alpha\pi_e\pi'' = \alpha\pi''_t$.

From the previous case distinction we can conclude that $\tau_t\pi_e\pi'' = \tau_t\pi''_t = \tau'_t$ (by C).

To prove $\tau_e\pi'' = \tau'_e$ consider a type variable $\alpha \in ftv(\tau_e)$:

- If $\alpha \in B$ then $\alpha \notin ftv((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$, so by Remark 1 α is a fresh variable and cannot appear in $dom(\pi''_t)$, so $\alpha \notin dom(\pi''_t|_B)$.
- If $\alpha \notin B$ then $\alpha \notin dom(\pi''_t|_B)$.

Therefore $\tau_e\pi'' = \tau_e(\pi''_t|_B + \pi''_e) = \tau_e\pi''_e = \tau'_e$ (by D).

Using the previous equalities it is easy to prove that:

$$\begin{array}{lll}
 & (\tau_t\pi_e \rightarrow_{()} anArgs(m, \tau_t\pi_e, \tau_e))\pi'' & \\
= & \tau_t\pi_e\pi'' \rightarrow_{()} anArgs(m, \tau_t\pi_e, \tau_e)\pi'' & \text{substitution application} \\
= & \tau_t\pi_e\pi'' \rightarrow_{()} anArgs(m\tau_t\pi_e\pi'', \tau_e\pi'') & \text{Lemma 6} \\
= & \tau'_t \rightarrow_{()} anArgs(m, \tau'_t, \tau'_e) & \text{previous equalities}
\end{array}$$

□

B.4 Proof of Lemma 1

Lemma 1. *If $\mathcal{A} \vdash \lambda\overline{t_{n+1}}.e : \tau$ then τ is n-transparent. Similarly, if $\mathcal{A} \Vdash \lambda\overline{t_{n+1}}.e : \tau'|\pi'$ then τ' is n-transparent.*

Proof. Easily from Lemmas 4 and 5, since $\lambda depth(\lambda\overline{t_{n+1}}.e) = n + 1$. □

B.5 Proof of Theorem 3 (Type Preservation)

To prove Theorem 3 we need the following result about the instantiation of variables in patterns:

Lemma 8. *Assume $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$, where $fv(t) \subseteq \{\overline{X_n}\}$. If $\mathcal{A} \vdash t[\overline{X_n \mapsto t_n}] : \tau$ then $\mathcal{A} \vdash t_i : \tau_i$ for all $X_i \in fv(t)$.*

Proof. By induction over the structure of t:

BASE CASE:

- $t \equiv X \in \mathcal{DV}$
  Straightforward.

INDUCTIVE STEP:

- $t \equiv c\,\overline{t'_m}$ with $m \leq ar(c)$
  We have the type derivation:

$$\cfrac{\cfrac{\cfrac{}{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash c : \tau'_1 \rightarrow_{\rho'_1} \ldots \rightarrow_{\rho'_{m-1}} \tau'_m \rightarrow_{\rho'_m} \tau'}\,(\text{ID}) \\ \vdots}{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash c\,\overline{t'_{m-1}} : \tau'_m \rightarrow_{\rho'_m} \tau'}\,(\text{APP}) \qquad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t'_m : \tau'_m}{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash c\,\overline{t'_m} : \tau'}\,(\text{APP})$$

On the other hand, denoting $[\overline{X_n \mapsto t_n}]$ as θ the type derivation $\mathcal{A} \vdash t\theta : \tau'$ is:

$$\cfrac{\cfrac{\cfrac{}{\mathcal{A} \vdash c : \tau''_1 \rightarrow_{\rho''_1} \ldots \rightarrow_{\rho''_{m-1}} \tau''_m \rightarrow_{\rho''_m} \tau''}\,(\text{ID}) \\ \vdots}{\mathcal{A} \vdash c\,\overline{t'_{m-1}}\theta : \tau''_m \rightarrow_{\rho''_m} \tau''}\,(\text{APP}) \qquad \mathcal{A} \vdash t'_m\theta : \tau''_m}{\mathcal{A} \vdash c\,\overline{t'_m}\theta : \tau''}\,(\text{APP})$$

where $\tau'' \equiv \tau'$. By definition of set of assumptions we know that $\mathcal{A}(c)$ is m-transparent, i.e., $\mathcal{A}(c) = \forall\overline{\alpha_k}.\tau_1 \rightarrow_{\rho_1} \ldots \rightarrow_{\rho_{m-1}} \tau_m \rightarrow_{\rho_m} \tau$ and $ftv(\tau_1 \rightarrow_{\rho_1} \ldots \rightarrow_{\rho_{m-1}} \tau_m) \subseteq ftv(\tau)$. The types derived for the constructor symbol c in both derivations are generic instances of $\mathcal{A}(c)$, so

$\tau'_1 \rightarrow_{\rho'_1} \ldots \rightarrow_{\rho'_{m-1}} \tau'_m \rightarrow_{\rho'_m} \tau' \equiv (\tau_1 \rightarrow_{\rho_1} \ldots \rightarrow_{\rho_{m-1}} \tau_m \rightarrow_{\rho_m} \tau)\pi_1$
$\tau''_1 \rightarrow_{\rho''_1} \ldots \rightarrow_{\rho''_{m-1}} \tau''_m \rightarrow_{\rho''_m} \tau'' \equiv (\tau_1 \rightarrow_{\rho_1} \ldots \rightarrow_{\rho_{m-1}} \tau_m \rightarrow_{\rho_m} \tau)\pi_2$
for some type substitutions π_1 and π_2. From $\tau'' \equiv \tau'$ we have $\tau\pi_1 \equiv \tau\pi_2$, so by transparency of $\mathcal{A}(c)$—$ftv(\tau_1 \rightarrow_{\rho_1} \ldots \rightarrow_{\rho_{n-1}} \tau_n) \subseteq ftv(\tau)$—we know that $\tau'_j \equiv \tau_j\pi_1 \equiv \tau_j\pi_2 \equiv \tau''_j$ for $j \in [1..m]$. Since $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t'_j : \tau'_j$, $\mathcal{A} \vdash t'_j[\overline{X_n \mapsto t_n}] : \tau'_j$ and $fv(t'_j) \subseteq fv(t) \subseteq \{\overline{X_n}\}$ then by the Induction Hypothesis for every $j \in [1..m]$ $\mathcal{A} \vdash t_i : \tau_i$ for all $X_i \in fv(t'_j)$. Collecting all these results, we have that $\mathcal{A} \vdash t_i : \tau_i$ for all $X_i \in fv(t)$ (as any $X_i \in fv(t)$ will appear in some subpattern t'_j).

- $t \equiv f\ \overline{t_m}$ with $m < ar(f)$
  Similar to the previous case, considering that by definition $\mathcal{A}(f)$ is $(m-1)$-transparent.

□

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

Proof. By a case distinction over the let-rewriting rule used in the step $e \rightarrow^l e'$. The proofs for the cases (LetIn), (Bind), (Elim), (Flat), (LetAp) and (Contx) are the same as in [20]—notice that let$_m$-expressions in [20] and let-expressions in this paper have the same type—so we only detail the remaining cases:

(Fapp) We consider a function that accepts two arguments, but the proof for any other number of arguments follows the same ideas. If we reduce an expression e using the (Fapp) rule then the step is $e \equiv f\ t_1\theta\ t_2\theta \rightarrow^l r\theta$ (being $f\ t_1\ t_2 \rightarrow r$ a rule in $\mathcal{P}$ and $\theta \in \mathcal{PS}ubst$). In this case we want to prove that $\mathcal{A} \vdash r\theta : \tau$. Since $wt_{\mathcal{A}}(\mathcal{P})$, then $\mathcal{A} \vdash \lambda t_1.\lambda t_2.r : \tau'_1 \rightarrow \tau'_2 \rightarrow \tau'_r$, being $\tau'_1 \rightarrow \tau'_2 \rightarrow \tau'_r$ a variant of $\mathcal{A}(f)$. We assume that the variables of the patterns t_1 and t_2 do not appear in $\mathcal{A}$ or in $vran(\theta)$. Then the tree for this type derivation will be:

$$\textbf{[}\Lambda\textbf{]}\ \dfrac{\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \vdash t_1 : \tau'_1 \qquad \textbf{[}\Lambda\textbf{]}\ \dfrac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \oplus \{\overline{Y_m : \tau''_{2_m}}\} \vdash t_2 : \tau'_2 \\ \mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \oplus \{\overline{Y_m : \tau''_{2_m}}\} \vdash r : \tau'\end{array}}{\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \vdash \lambda t_2.r : \tau'_2 \rightarrow \tau'}}{\mathcal{A} \vdash \lambda t_1.\lambda t_2.r : \tau'_1 \rightarrow \tau'_2 \rightarrow \tau'}$$

As variables $\overline{X_n}$ and $\overline{Y_m}$ are all different (the left hand side of the rules is linear), by Lemma 7-b we can add the assumptions over $\overline{Y_m}$ to the derivation of t_1, obtaining $\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \oplus \{\overline{Y_m : \tau''_{2_m}}\} \vdash t_1 : \tau'_1$.
It is a premise that $\mathcal{A} \vdash f\ t_1\theta\ t_2\theta : \tau$, so the type derivation will be:

$$\textbf{[APP]}\ \dfrac{\textbf{[APP]}\ \dfrac{\begin{array}{c}\mathcal{A} \vdash f : \tau_1 \rightarrow \tau_2 \rightarrow \tau \\ \mathcal{A} \vdash t_1\theta : \tau_1\end{array}}{\mathcal{A} \vdash f\ t_1 : \tau_2 \rightarrow \tau} \qquad \mathcal{A} \vdash t_2\theta : \tau_2}{\mathcal{A} \vdash f\ t_1\theta\ t_2\theta : \tau}$$

Therefore we know that (A) $\mathcal{A} \vdash t_1\theta : \tau_1$, $\mathcal{A} \vdash t_2\theta : \tau_2$ and $\mathcal{A} \vdash f : \tau_1 \rightarrow \tau_2 \rightarrow \tau$, being $\tau_1 \rightarrow \tau_2 \rightarrow \tau$ a generic instance of the type $\mathcal{A}(f)$. Then there will exist a type substitution π such that $\tau'_1\pi = \tau_1$, $\tau'_2\pi = \tau_2$ and $\tau'\pi = \tau$. Moreover, $dom(\pi)$ does not contain any free type variable in $\mathcal{A}$, since π transforms a variant of the type $\mathcal{A}(f)$ into a generic instance of $\mathcal{A}(f)$. Then by Lemma 7-a we have

$$(\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \oplus \{\overline{Y_m : \tau''_{2_m}}\})\pi \vdash t_2 : \tau'_2\pi$$

$$(\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\}) \oplus \{\overline{Y_m : \tau''_{2_m}}\})\pi \vdash t_1 : \tau'_1\pi$$

which is equal to

$$(B)\ \mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}\pi}\} \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash t_2 : \tau_2$$

$$(B)\ \mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}\pi}\}) \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash t_1 : \tau_1$$

With (A) and (B) and by Lemma 8 we can state that for every variable X_i in r (remember that $X_i \in fv(t_1) \cup fv(t_2)$ by linearity of program rules) then $\mathcal{A} \vdash X_i\theta : \tau''_{1_i}\pi$ (resp. $\mathcal{A} \vdash Y_j\theta : \tau''_{2_i}\pi$). None of the variables $\overline{X_n}$, $\overline{Y_m}$ appear in $X_i\theta$ or $Y_j\theta$, so by Lemma 7-b we can add these assumptions and obtain

$$(D)\ \mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}\pi}\} \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash X_i\theta : \tau''_{1_i}\pi$$

$$(D)\ \mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}\pi}\} \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash Y_j\theta : \tau''_{2_i}\pi$$

Applying Lemma 7-a to the derivation of r we obtain

$$\mathcal{A}\pi \oplus \{\overline{X_n : \tau''_{1_n}\pi}\} \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash r : \tau'\pi$$

Using (D) and Lemma 7-c we can replace data variables by expressions of the same type:

$$\mathcal{A}\pi \oplus \{\overline{X_n : \tau''_{1_n}\pi}\} \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash r\theta : \tau'\pi$$

Since no variable $\overline{X_n}$, $\overline{Y_m}$ appear in $r\theta$ by Lemma 7-b we can remove their assumptions, obtaining a derivation $\mathcal{A}\pi \vdash r\theta : \tau'\pi$. Finally, using the fact that $\mathcal{A}\pi = \mathcal{A}$ and $\tau'\pi = \tau$, this last derivation is equal to $\mathcal{A} \vdash r\theta : \tau$.

(JoinS) Straightforward since $\mathcal{A} \vdash s \bowtie s : bool$ and $\mathcal{A} \vdash true : bool$.

(JoinP) We treat the case of $n = 2$, but the proof for any other number of arguments ($n > 0$) follows the same ideas. In this case we have the step $(h\ t_1\ t_2) \bowtie (h\ t'_1\ t'_2) \to^l (t_1 \bowtie t'_1) \wedge (t_2 \bowtie t'_2)$ and the type derivation:

$$\text{(APP)}\dfrac{\text{(APP)}\dfrac{\mathcal{A} \vdash (\bowtie) : \tau \to_{()} \tau \to_{(\tau)} bool \qquad \mathcal{A} \vdash h\ t_1\ t_2 : \tau}{\mathcal{A} \vdash (\bowtie)\ (h\ t_1\ t_2) : \tau \to_{(\tau)} bool} \qquad \mathcal{A} \vdash h\ t'_1\ t'_2 : \tau}{\mathcal{A} \vdash (\bowtie)\ (h\ t_1\ t_2)\ (h\ t'_1\ t'_2)}$$

(notice that we use $\bowtie$ as a prefix to adequate it to the typing rules). Both $h\ t_1\ t_2$ and $h\ t'_1\ t'_2$ have the same type τ. Since h is transparent we know that $\mathcal{A}(h) = \forall\overline{\alpha}.\tau'_1 \to_{\rho_1} \tau'_2 \to_{\rho_2} \tau'$, where $ftv(\tau'_1 \to_{\rho_1} \tau'_2) \subseteq ftv(\tau')$. In the derivations $\mathcal{A} \vdash h\ t_1\ t_2 : \tau$ and $\mathcal{A} \vdash h\ t'_1\ t'_2 : \tau$ there will be sub-derivations $\mathcal{A} \vdash h : \tau''_1 \to_{\rho''_1} \tau''_2 \to_{\rho''_2} \tau$ and $\mathcal{A} \vdash h : \tau'''_1 \to_{\rho'''_1} \tau'''_2 \to_{\rho'''_2} \tau$ respectively. These types are generic instances of $\mathcal{A}(h)$, so there are substitutions $\pi_1 \equiv [\overline{\alpha \mapsto \tau^1}]$ and $\pi_2 \equiv [\overline{\alpha \mapsto \tau^2}]$ such that:

$$(\tau'_1 \to_{\rho_1} \tau'_2 \to_{\rho_2} \tau')\pi_1 \equiv \tau''_1 \to_{\rho''_1} \tau''_2 \to_{\rho''_2} \tau$$
$$(\tau'_1 \to_{\rho_1} \tau'_2 \to_{\rho_2} \tau')\pi_2 \equiv \tau'''_1 \to_{\rho'''_1} \tau'''_2 \to_{\rho'''_2} \tau$$

Since $\tau'\pi_1 \equiv \tau \equiv \tau'\pi_2$ and $ftv(\tau'_1 \to_{\rho_1} \tau'_2) \subseteq ftv(\tau')$ then $\tau''_1 \equiv \tau'_1\pi_1 \equiv \tau'_1\pi_2 \equiv \tau'''_1$ and $\tau''_2 \equiv \tau'_2\pi_1 \equiv \tau'_2\pi_2 \equiv \tau'''_2$. Therefore t_1 and t'_1 have the same type, as well as t_2 and t'_2, so $\mathcal{A} \vdash t_1 \bowtie t'_1 : bool$ and $\mathcal{A} \vdash t_2 \bowtie t'_2 : bool$. Then we have the type derivation:

$$\text{(APP)}\dfrac{\text{(APP)}\dfrac{\mathcal{A} \vdash (\wedge) : bool \to_{()} bool \to_{(bool)} bool \qquad \mathcal{A} \vdash (t_1 \bowtie t'_1) : bool}{\mathcal{A} \vdash (\wedge)\ (t_1 \bowtie t'_1) : bool \to_{(bool)} bool} \qquad \mathcal{A} \vdash t_2 \bowtie t'_2 : bool}{\mathcal{A} \vdash (\wedge)\ (t_1 \bowtie t'_1)\ (t_2 \bowtie t'_2) : bool}$$

where $(\wedge)$ is treated as a prefix.

□

B.6 Proof of Theorem 4

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

Proof. Easy, because as $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ then by Theorem 1 we have $\mathcal{A}\pi \vdash (\varphi(R_1), \ldots, \varphi(R_m)) : (\tau_1, \ldots, \tau_m)$. Because of the type of *pair* and point 2 of $\mathcal{B}$ (Definition 3) it is clear that each τ_i is a variant of $\mathcal{A}\pi(f_i)$. □

B.7 Proof for Theorem 5

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

Proof. Since $wt_{\mathcal{A}\pi'}(\mathcal{P})$ we know that for every rule $R_i \equiv f_i\ t_1 \ldots t_n \to e_i$ in $\mathcal{P}$ there exists a type derivation $\mathcal{A}\pi' \vdash \lambda t_1 \ldots t_n.e_i : \tau'_i$ and τ'_i is a variant of the type $\mathcal{A}\pi'(f_i)$, so $\mathcal{A}\pi' \vdash f_i : \tau'_i$. With these derivations we can build $\mathcal{A}\pi' \vdash (\varphi(R_1), \ldots, \varphi(R_m)) : (\tau'_1, \ldots, \tau'_m)$. By Theorem 2 we know that $\mathcal{A} \Vdash (\varphi(R_1), \ldots, \varphi(R_m)) : (\tau_1, \ldots, \tau_m)|\pi$, that is the first point of $\mathcal{B}$, and there is some type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $(\tau_1, \ldots, \tau_m)\pi'' \equiv (\tau'_1, \ldots, \tau'_m)$. From $wt_{\mathcal{A}\pi'}(\mathcal{P})$ we know that for every rule $f\ \overline{t_n} \to e$ of a function symbol f^i with a closed type-scheme as assumption in $\mathcal{A}\pi'$, $\mathcal{A}\pi'(f^i) \succ_{var} \tau_i\pi'' \equiv \tau'_i$. Therefore it is easy to see that τ_i must be also a variant of $\mathcal{A}\pi'(f^i) = \mathcal{A}\pi(f^i) \equiv \mathcal{A}(f^i)$, which is the second point of $\mathcal{B}$. Therefore $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ and $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ for some π''. □

B.8 Proof for Lemma 2

Lemma 2. *If $\mathcal{A} \vdash^v \lambda\overline{t_{n+1}}.e : \tau$ then τ is n-transparent.*

Proof. Considering the rule (Λ^v) (Figure 5) and $anArgs^v$ (Definition 4) the type derivation will be:

$$\mathcal{A} \vdash^v \lambda\overline{t_{n+1}}.e : \tau_1 \to_{()} \tau_2 \to_{\chi_1} \ldots \to_{\chi_{n-2}} \tau_n \to_{\chi_{n-1}} \tau_{n+1} \to_{\chi_n} \tau'$$

where χ_i is a sequence that contains the type variables in $\tau_1 \ldots \tau_i$. This type is clearly n-transparent because $ftv(\tau_1 \to_{()} \tau_2 \to_{\chi_1} \ldots \to_{\chi_{n-2}} \tau_n) \subseteq ftv(\tau_{n+1} \to_{\chi_n} \tau')$ as χ_n contains all the type variables in the $\tau_1 \ldots \tau_n$. □

B.9 Proof for Theorem 7 (Type Preservation)

It is important to note that $\vdash$ and $\vdash^v$ are equivalent type relation when considering expressions without λ-abstractions:

Lemma 9. *If e is an expression not containing λ-abstractions then $\mathcal{A} \vdash e : \tau$ iff $\mathcal{A} \vdash^v e : \tau$.*

Proof. Straightforward since the typing rules for symbols, applications and expression are the same in both type relations. □

In order to prove type preservation for well-typed programs using $\vdash^v$ (Definition 5) we need to some properties of this type relation. These properties are the same as the ones for $\vdash$ stated in Lemma 7 but limited to expressions not containing λ-abstractions. Notice that although $\vdash^v$ is not closed under type substitutions for general expressions, it is closed under type substitutions if we only consider expressions without λ-abstractions.

Lemma 10 (Properties of $\vdash^v$).
If e and e' are expressions not containing λ-abstractions then:

a) If $\mathcal{A} \vdash^v e : \tau$ then $\mathcal{A}\pi \vdash^v e : \tau\pi$.
b) Let s be a symbol not appearing in e, and σ_s any type-scheme. Then $\mathcal{A} \vdash^v e : \tau \Leftrightarrow \mathcal{A} \oplus \{s : \sigma_s\} \vdash^v e : \tau$.
c) Assume $\mathcal{A}(X) = \tau_x$ and $\mathcal{A} \vdash^v e' : \tau_x$. If $\mathcal{A} \vdash^v e : \tau$ then $\mathcal{A} \vdash^v e[X \mapsto e'] : \tau$.

Proof. Easily using Lemma 9. □

We also need a result similar to Lemma 8 but considering the type relation $\vdash^v$:

Lemma 11. *Assume $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash^v t : \tau$, where $fv(t) \subseteq \{\overline{X_n}\}$. If $\mathcal{A} \vdash^v t[\overline{X_n \mapsto t_n}] : \tau$ then $\mathcal{A} \vdash^v t_i : \tau_i$ for all $X_i \in fv(t)$.*

Proof. The proof is the same as in Lemma 8, as the typing of patterns in $\vdash$ and $\vdash^v$ is the same by Lemma 9. The only difference is that $wt^v_{\mathcal{A}}(\mathcal{P})$ forces that arrow decorations in assumed types for function symbols in $\mathcal{P}$ must be sequences of type variables instead of sequences of simple types. However those assumptions are still transparent, which is the key property used in the proof. □

Theorem 7 (Type Preservation). *If $wt^v_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash^v e : \tau$ and $e \to^l e'$ then $\mathcal{A} \vdash^v e' : \tau$.*

Proof. The proof is similar to the proof of type preservation using $\vdash$ (Lemma 3) with the following exceptions:

- (Fapp) case uses Lemma 11 instead of Lemma 8.
- All the cases use the properties of $\vdash^v$ stated in Lemma 10, that are always applied to expressions not containing λ-abstractions (remember that these expressions can only come from the program or are expressions to evaluate).

□

B.10 Proof of Theorem 8

To prove Theorem 8 we need the following auxiliary results:

Lemma 12. $\mathit{flat}(\mathit{anArgs}^\bullet(n, \tau_1^\bullet, \tau_2^\bullet)) = \mathit{anArgs}^v(n, \mathit{flat}(\tau_1^\bullet), \mathit{flat}(\tau_2^\bullet))$

Proof. By induction on n:
BASE CASE: $\mathit{flat}(\mathit{anArgs}^\bullet(0, \tau_1^\bullet, \tau_2^\bullet)) = \mathit{flat}(\tau_2^\bullet) = \mathit{anArgs}^v(0, \mathit{flat}(\tau_1^\bullet), \mathit{flat}(\tau_2^\bullet))$.
INDUCTIVE STEP: In this case we know that $\tau_2^\bullet \equiv \tau_a^\bullet \to_{\bullet(\overline{\tau^\bullet})} \tau_b^\bullet$, otherwise $\mathit{anArgs}^\bullet$ and anArgs^v are not defined. Then $\mathit{flat}(\tau_a^\bullet \to_{\bullet(\overline{\tau^\bullet})} \tau_b^\bullet) \equiv \mathit{flat}(\tau_a^\bullet) \to_\chi \mathit{flat}(\tau_b^\bullet)$, where χ is the sequence of type variables occurring in $\overline{\tau^\bullet}$. Therefore:

$$\begin{array}{cll}
 & \mathit{flat}(\mathit{anArgs}^\bullet(n, \tau_1^\bullet, \tau_2^\bullet)) & \\
= & \mathit{flat}(\mathit{anArgs}^\bullet(n, \tau_1^\bullet, \tau_a^\bullet \to_{\bullet(\overline{\tau^\bullet})} \tau_b^\bullet)) & \\
= & \mathit{flat}(\tau_a^\bullet \to_{\bullet(\overline{\tau_1^\bullet, \tau^\bullet})} \mathit{anArgs}^\bullet(n-1, \tau_1^\bullet, \tau_b^\bullet)) & \text{definition of } \mathit{anArgs}^\bullet \\
= & \mathit{flat}(\tau_a^\bullet) \to_{\chi' \diamond \chi} \mathit{flat}(\mathit{anArgs}^\bullet(n-1, \tau_1^\bullet, \tau_b^\bullet)) & \text{definition of } \mathit{flat} \\
= & \mathit{flat}(\tau_a^\bullet) \to_{\chi' \diamond \chi} \mathit{anArgs}^v(n-1, \mathit{flat}(\tau_1^\bullet), \mathit{flat}(\tau_b^\bullet)) & \text{by (IH)} \\
= & \mathit{anArgs}^v(n, \mathit{flat}(\tau_1^\bullet), \mathit{flat}(\tau_a^\bullet) \to_{(\chi)} \mathit{flat}(\tau_b^\bullet)) & \text{definition of } \mathit{anArgs}^v \\
= & \mathit{anArgs}^v(n, \mathit{flat}(\tau_1^\bullet), \mathit{flat}(\tau_2^\bullet)) &
\end{array}$$

where χ' are the sequence of type variables appearing in $\tau_1^\bullet$, which is the same sequence of type variables occurring in $\mathit{flat}(\tau_1^\bullet)$. □

Lemma 13. *If $\mathcal{A} \oplus \{\overline{X_m : \tau_m^\bullet}\} \vdash^m e : \tau^\bullet$ then $\mathcal{A} \oplus \{\overline{X_m : \mathit{flat}(\tau_m^\bullet)}\} \vdash^v e : \mathit{flat}(\tau^\bullet)$.*

Proof. By induction on the structure of e.
BASE CASE:

- $e \equiv s$) Straightforward by case distinction whether s is a type variable in $\{\overline{X_m}\}$ or not.

  INDUCTIVE STEP: Easy, we only include the most interesting case: λ-abstractions.

- $e \equiv \lambda t.e$) We have the type derivation

$$(\Lambda^\bullet)\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_m : \tau_m^\bullet}\} \oplus \{\overline{Z_n : \tau_n'^\bullet}\} \vdash^m t : \tau_t^\bullet \\ \mathcal{A} \oplus \{\overline{X_m : \tau_m^\bullet}\} \oplus \{\overline{Z_n : \tau_n'^\bullet}\} \vdash^m e : \tau^\bullet\end{array}}{\mathcal{A} \oplus \{\overline{X_m : \tau_m^\bullet}\} \vdash^m \lambda t.e : \tau_t^\bullet \to_{\bullet()} \mathit{anArgs}^\bullet(k, \tau_t^\bullet, \tau^\bullet)}$$

  where $\{\overline{Z_n}\} = var(t)$ and $k = \lambda depth(\lambda t.e)$. By the Induction Hypothesis we have:

$$\begin{array}{c}\mathcal{A} \oplus \{\overline{X_m : \mathit{flat}(\tau_m^\bullet)}\} \oplus \{\overline{Z_n : \mathit{flat}(\tau_n'^\bullet)}\} \vdash^v t : \mathit{flat}(\tau_t^\bullet) \\ \mathcal{A} \oplus \{\overline{X_m : \mathit{flat}(\tau_m^\bullet)}\} \oplus \{\overline{Z_n : \mathit{flat}(\tau_n'^\bullet)}\} \vdash^v e : \mathit{flat}(\tau^\bullet)\end{array}$$

so we have the type derivation:

$$(\Lambda^v)\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_m : \mathit{flat}(\tau_m^\bullet)}\} \oplus \{\overline{Z_n : \mathit{flat}(\tau_n'^\bullet)}\} \vdash^v t : \mathit{flat}(\tau_t^\bullet) \\ \mathcal{A} \oplus \{\overline{X_m : \mathit{flat}(\tau_m^\bullet)}\} \oplus \{\overline{Z_n : \mathit{flat}(\tau_n'^\bullet)}\} \vdash^v e : \mathit{flat}(\tau^\bullet)\end{array}}{\mathcal{A} \oplus \{\overline{X_m : \mathit{flat}(\tau_m^\bullet)}\} \vdash^v \lambda t.e : \mathit{flat}(\tau_t^\bullet) \to_{()} \mathit{anArgs}^v(k, \mathit{flat}(\tau_t^\bullet), \mathit{flat}(\tau^\bullet))}$$

Using Lemma 12 it is easy to check that $\mathit{flat}(\tau_t^\bullet \to_{\bullet()} \mathit{anArgs}^\bullet(k, \tau_t^\bullet, \tau^\bullet)) = \mathit{flat}(\tau_t^\bullet) \to_{()} \mathit{anArgs}^v(k, \mathit{flat}(\tau_t^\bullet), \mathit{flat}(\tau^\bullet))$.

□

Lemma 14. *If $\mathcal{A} \oplus \{\overline{X_m : \tau_m}\} \vdash^v e : \tau$ then there are some $\overline{\tau_m^\bullet}$ such that $\mathcal{A} \oplus \{\overline{X_m : \tau_m^\bullet}\} \vdash^m e : \tau^\bullet$ and $\mathit{flat}(\tau^\bullet) = \tau$.*

Proof. By induction on the structure of e.
BASE CASE:

- $e \equiv s$) Straightforward by case distinction whether s is a type variable in $\{\overline{X_m}\}$ or not.

INDUCTIVE STEP: Easily by induction on the structure of e. As before, we only include the case for λ-abstractions:

- $e \equiv \lambda t.e$) We have the type derivations

$$(\Lambda^v)\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash^v t : \tau_t \\ \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash^v e : \tau\end{array}}{\mathcal{A} \vdash^v \lambda t.e : \tau_t \to_{()} \mathit{anArgs}^v(k, \tau_t, \tau)}$$

where $\{\overline{X_n}\} = \mathit{var}(t)$ and $k = \lambda\mathit{depth}(\lambda t.e)$. By the Induction Hypothesis we have:

$$\mathcal{A} \oplus \{\overline{X_n : \tau_n^\bullet}\} \vdash^m t : \tau_t^\bullet \text{ and } \mathit{flat}(\tau_t^\bullet) = \tau_t$$
$$\mathcal{A} \oplus \{\overline{X_n : \tau_n^\bullet}\} \vdash^m e : \tau^\bullet \text{ and } \mathit{flat}(\tau^\bullet) = \tau$$

Therefore we can build the type derivation:

$$(\Lambda^v)\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \tau_n^\bullet}\} \vdash^m t : \tau_t^\bullet \\ \mathcal{A} \oplus \{\overline{X_n : \tau_n^\bullet}\} \vdash^m e : \tau^\bullet\end{array}}{\mathcal{A} \vdash^v \lambda t.e : \tau_t^\bullet \to_{\bullet()} \mathit{anArgs}^\bullet(k, \tau_t^\bullet, \tau^\bullet)}$$

Finally, using Lemma 12 it is easy to check that

$$\begin{array}{cl} & \mathit{flat}(\tau_t^\bullet \to_{\bullet()} \mathit{anArgs}^\bullet(k, \tau_t^\bullet, \tau^\bullet)) \\ = & \mathit{flat}(\tau_t^\bullet) \to_{()} \mathit{flat}(\mathit{anArgs}^\bullet(k, \tau_t^\bullet, \tau^\bullet)) \\ = & \mathit{flat}(\tau_t^\bullet) \to_{()} \mathit{anArgs}^v(k, \mathit{flat}(\tau_t^\bullet), \mathit{flat}(\tau^\bullet)) \\ = & \tau_t \to_{()} \mathit{anArgs}^v(k, \tau_t, \tau) \end{array}$$

□

With the previous results, we can now prove Theorem 8.
Theorem 8. *$\mathcal{A} \vdash^m e : \tau^\bullet$ and $\tau = \mathit{flat}(\tau^\bullet)$ iff $\mathcal{A} \vdash^v e : \tau$.*

Proof.

- $\Longrightarrow$) This directions is proved directly by Lemma 13, as it is the particular case when $\{X_n\} = \emptyset$.
- $\Longleftarrow$) Using Lemma 14, as this is the particular case when $\{X_n\} = \emptyset$.

□

B.11 Proof of Lemma 3 (Closure of $\vdash^m$)

In order to prove that $\vdash^m$ is closed under type substitutions we need a result similar to Lemma 6 relating $\mathit{anArgs}^\bullet$ and marked type assumptions:

Lemma 15. *If $\mathit{anArgs}^\bullet(n, \tau^\bullet, \tau'^\bullet)$ is defined then $\mathit{anArgs}^\bullet(n, \tau^\bullet, \tau'^\bullet)\pi^\bullet = \mathit{anArgs}^\bullet(n, \tau^\bullet\pi^\bullet, \tau'^\bullet\pi^\bullet)$*

Proof. By induction on n. □

Lemma 3 (Closure of $\vdash^m$). *If $\mathcal{A} \vdash^m e : \tau^\bullet$ then $\mathcal{A}\pi^\bullet \vdash^m e : \tau^\bullet\pi^\bullet$.*

Proof. By induction over the structure of e, using Lemma 15 in the $(\Lambda^\bullet)$ case. □

B.12 Proof of Theorem 9 (Type Preservation)

Theorem 9 (Type Preservation). *If $wt^m_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash^m e : \tau$ and $e \to^l e'$ then $\mathcal{A} \vdash^m e' : \tau$.*

Proof. We know that $wt^m_{\mathcal{A}}(\mathcal{P})$ and $\mathcal{A} \vdash^m e : \tau$. From Theorem 8 follows easily that $wt^v_{\mathcal{A}}(\mathcal{P})$ iff $wt^m_{\mathcal{A}}(\mathcal{P})$, so we also know that $wt^v_{\mathcal{A}}(\mathcal{P})$. As $\vdash^v$ and $\vdash^m$ are equivalent for expressions not containing λ-abstractions when considering unmarked types, we have that $\mathcal{A} \vdash^v e : \tau$. Applying the type preservation property for well-typed programs using $\vdash^v$, we obtain that $\mathcal{A} \vdash^v e' : \tau$. Finally, as e' cannot contain λ-abstractions, by the same reasoning as before we have $\mathcal{A} \vdash^m e' : \tau$. $\square$

B.13 Proof of Theorem 10 (Properties of $\Vdash^m$ w.r.t. $\vdash^m$)

Theorem 10 (Properties of $\Vdash^m$ w.r.t. $\vdash^m$).

- *(Soundness)* If $\mathcal{A}^\bullet \Vdash^m e : \tau^\bullet | \pi^\bullet$ then $\mathcal{A}^\bullet\pi^\bullet \vdash^m e : \tau^\bullet$.
- *(Completeness)* If $\mathcal{A}\pi_1^\bullet \vdash^m e : \tau_1^\bullet$ then $\mathcal{A}^\bullet \Vdash^m e : \tau^\bullet | \pi^\bullet$ and there is some $\pi_2^\bullet$ verifying $\mathcal{A}^\bullet\pi^\bullet\pi_2^\bullet = \mathcal{A}_1^\bullet$ and $\tau^\bullet\pi_2^\bullet = \tau_1^\bullet$.

Proof.

- *(Soundness)* By induction over the type inference, using the closure under type substitution of $\vdash^m$ (Lemma 3).
- *(Completeness)* By induction over the type derivation. The proof is similar to the proof for completeness of $\Vdash$ (Theorem 2). $\square$

B.14 Proof of Theorem 11 (Soundness of $\mathcal{B}^\bullet$)

To prove the soundness of $\mathcal{B}^\bullet$ we need some auxiliary results:

Lemma 16. *Consider an expression e not containing λ-abstractions and a set of assumptions $\mathcal{A}$ containing closed type-schemes (therefore unmarked) and type variables. If $\mathcal{A}\pi^\bullet \vdash^m e : \tau^\bullet$ then $\mathcal{A}\pi' \vdash e : flat(\tau^\bullet)$, where $\pi' = flat(\pi^\bullet)$.*

Proof. By induction on the structure of e:

BASE CASE:

- $e \equiv s$) If $\mathcal{A}(s)$ is a closed type-scheme then $\mathcal{A}\pi^\bullet(s) = \mathcal{A}\pi'(s) = \forall\overline{\alpha_n}.\tau_s$. If $\mathcal{A}\pi^\bullet \vdash^m s : \tau^\bullet$ then $\mathcal{A}\pi^\bullet(s) \succ \tau^\bullet = \tau_s[\overline{\alpha_n \mapsto \tau_n^\bullet}]$. Therefore it is clear that $\mathcal{A}\pi' \vdash^m s : flat(\tau^\bullet)$ because $\mathcal{A}\pi'(s) \succ \tau_s[\overline{\alpha_n \mapsto flat(\tau_n^\bullet)}] = flat(\tau^\bullet)$. If $\mathcal{A}(s)$ is a type variable it follows directly from the premises.

INDUCTIVE STEP:

- $e \equiv e_1\ e_2$) We have the type derivation:

$$(\text{APP}^\bullet)\frac{\begin{array}{c}\mathcal{A}\pi^\bullet \vdash^m e_1 : \tau_1^\bullet \to_{\rho^\bullet} \tau_2^\bullet \\ \mathcal{A}\pi^\bullet \vdash^m e_2 : \tau_1^\bullet\end{array}}{\mathcal{A}\pi^\bullet \vdash^m e_1\ e_2 : \tau_2^\bullet}$$

By the Induction Hypothesis we have $\mathcal{A}\pi' \vdash^m e_1 : flat(\tau_1^\bullet \to_{\rho^\bullet} \tau_2^\bullet)$ and $\mathcal{A}\pi' \vdash^m e_2 : flat(\tau_1^\bullet)$. As $flat(\tau_1^\bullet \to_{\rho^\bullet} \tau_2^\bullet) = flat(\tau_1^\bullet) \to_\chi flat(\tau_2^\bullet)$, where χ is the sequence of type variables in $\rho^\bullet$, we can build the type derivation:

$$(\text{APP}^\bullet)\frac{\begin{array}{c}\mathcal{A}\pi' \vdash^m e_1 : flat(\tau_1^\bullet) \to_\chi flat(\tau_2^\bullet) \\ \mathcal{A}\pi' \vdash^m e_2 : flat(\tau_1^\bullet)\end{array}}{\mathcal{A}\pi' \vdash^m e_1\ e_2 : flat(\tau_2^\bullet)}$$

- $e \equiv let\ X = e_1\ in\ e_2$) We have the type derivation:

$$(\text{LET}^\bullet)\frac{\begin{array}{c}\mathcal{A}\pi^\bullet \vdash^m e_1 : \tau_x^\bullet \\ \mathcal{A}\pi^\bullet \oplus \{X : \tau_x^\bullet\} \vdash^m e_2 : \tau^\bullet\end{array}}{\mathcal{A}\pi^\bullet \vdash^m let\ X = e_1\ in\ e_2 : \tau^\bullet}$$

By the Induction Hypothesis we know that $\mathcal{A}\pi' \vdash^m e_1 : flat(\tau_x^\bullet)$. The second derivation can be written as $(\mathcal{A}\oplus\{X : \alpha\})\pi_2^\bullet \vdash^m e_2 : \tau^\bullet$, where α is a fresh type variable and $\pi_2^\bullet \equiv \pi^\bullet \uplus [\alpha \mapsto \tau_x^\bullet]$. Then by the Induction Hyphothesis, obtaining $(\mathcal{A} \oplus \{X : \alpha\})\pi'' \vdash^m e_2 : flat(\tau^\bullet)$, where $\pi'' = flat(\pi_2^\bullet) = \pi' \uplus [\alpha \mapsto flat(\tau_x^\bullet)]$. The previous type derivation can then be written as $(\mathcal{A}\pi' \oplus (\{X : \alpha\})[\alpha \mapsto flat(\tau_x^\bullet)] \vdash^m e_2 : flat(\tau^\bullet)$. Therefore we can build the type derivation:

$$(\text{LET}^\bullet)\frac{\begin{array}{c}\mathcal{A}\pi' \vdash^m e_1 : flat(\tau_x^\bullet) \\ \mathcal{A}\pi' \oplus \{X : flat(\tau_x^\bullet)\} \vdash^m e_2 : flat(\tau^\bullet)\end{array}}{\mathcal{A}\pi^\bullet \vdash^m let\ X = e_1\ in\ e_2 : flat(\tau^\bullet)}$$

$\square$

Lemma 17. *If $flat(\tau^\bullet) = flat(\tau'^\bullet)$ then $flat(anArgs^\bullet(k, \tau_t^\bullet, \tau^\bullet)) = flat(anArgs^\bullet(k, flat(\tau_t^\bullet), \tau'^\bullet))$*

Proof. Easily by induction on k. $\square$

Lemma 18. *Consider a set of assumptions $\mathcal{A}$ containing closed type-schemes (therefore unmarked) and type variables. If $\mathcal{A}\pi^\bullet \vdash^m \lambda\overline{t_n}.e : \tau^\bullet$ such that e does not contain λ-abstractions and $\pi = flat(\pi^\bullet)$ then $\mathcal{A}\pi \vdash^m \lambda\overline{t_n}.e : \tau'^\bullet$ and $flat(\tau^\bullet) = flat(\tau'^\bullet)$.*

Proof. By induction on n. BASE CASE:

- $n = 0$) As e does not contains λ-abstractions by Lemma 16 then $\mathcal{A}\pi \vdash^m e : \mathit{flat}(\tau^\bullet)$. Clearly, $\mathit{flat}(\mathit{flat}(\tau^\bullet)) = \mathit{flat}(\tau^\bullet)$.

  INDUCTIVE STEP:

- $n > 0$) We have the type derivation:

$$(\Lambda^\bullet)\frac{\begin{array}{c}\mathcal{A}\pi^\bullet \oplus \{\overline{X_n : \tau_n^\bullet}\} \vdash t : \tau_t^\bullet \\ \mathcal{A}\pi^\bullet \oplus \{\overline{X_n : \tau_n^\bullet}\} \vdash \lambda\overline{t_{n-1}}.e : \tau^\bullet\end{array}}{\mathcal{A}\pi^\bullet \vdash^m \lambda\overline{t_n}.e : \tau_t^\bullet \rightarrow_{\bullet()} \mathit{anArgs}^\bullet(k, \tau_t^\bullet, \tau^\bullet)}$$

where $\{\overline{X_n}\} = \mathit{var}(t)$ and $k = \lambda\mathit{depth}(\lambda\overline{t_{n-1}}.e)$. The first type derivation can be written as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'^\bullet \vdash t : \tau_t^\bullet$, where $\pi'^\bullet = \pi^\bullet \uplus [\overline{\alpha_n \mapsto \tau_n^\bullet}]$ and $\overline{\alpha_n}$ are fresh type variables. As t is a pattern and it cannot contain λ-abstractions then by Lemma 16 we have that $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'' \vdash t : \mathit{flat}(\tau_t^\bullet)$, where $\pi'' = \mathit{flat}(\pi'^\bullet) = \pi \uplus [\overline{\alpha_n \mapsto \mathit{flat}(\tau_n^\bullet)}]$.
On the other hand the second type derivation can be written as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'^\bullet \vdash \lambda\overline{t_{n-1}}.e : \tau^\bullet$, so by the Induction Hypothesis we have that $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'' \vdash \lambda\overline{t_{n-1}}.e : \tau'^\bullet$ such that $\mathit{flat}(\tau^\bullet) = \mathit{flat}(\tau'^\bullet)$. Then we can build the following type derivation:

$$(\Lambda^\bullet)\frac{\begin{array}{c}\mathcal{A}\pi \oplus \{\overline{X_n : \tau_n^\bullet}\} \vdash t : \mathit{flat}(\tau_t^\bullet) \\ \mathcal{A}\pi \oplus \{\overline{X_n : \tau_n^\bullet}\} \vdash \lambda\overline{t_{n-1}}.e : \tau'^\bullet\end{array}}{\mathcal{A}\pi \vdash^m \lambda\overline{t_n}.e : \mathit{flat}(\tau_t^\bullet) \rightarrow_{\bullet()} \mathit{anArgs}^\bullet(k, \tau_t^\bullet, \tau'^\bullet)}$$

The point that remains to prove is

$$\mathit{flat}(\tau_t^\bullet \rightarrow_{\bullet()} \mathit{anArgs}^\bullet(k, \tau_t^\bullet, \tau^\bullet)) = \mathit{flat}(\mathit{flat}(\tau_t^\bullet) \rightarrow_{\bullet()} \mathit{anArgs}^\bullet(k, \mathit{flat}(\tau_t^\bullet), \tau'^\bullet)$$

This is clear by definition of $\mathit{anArgs}^\bullet$ and using Lemma 17, since $\mathit{flat}(\mathit{anArgs}^\bullet(k, \tau_t^\bullet, \tau^\bullet)) = \mathit{flat}(\mathit{anArgs}^\bullet(k, \mathit{flat}(\tau_t^\bullet), \tau'^\bullet))$ provided $\mathit{flat}(\tau^\bullet) = \mathit{flat}(\tau'^\bullet)$.

□

Theorem 11 (Soundness of $\mathcal{B}$). *Consider that $\mathcal{A}$ contains only closed type-schemes and type variables as assumptions for function symbols defined in $\mathcal{P}$. If $\mathcal{B}^\bullet(\mathcal{A}, \mathcal{P}) = \pi^\bullet$ and $\pi' = \mathit{flat}(\pi^\bullet)$ then $wt^m_{\mathcal{A}\pi'}(\mathcal{P})$.*

Proof. If $\mathcal{B}^\bullet(\mathcal{A}, \mathcal{P}) = \pi^\bullet$ then $\mathcal{A} \Vdash^m (\varphi(R_1), \dots, \varphi(R_m)) : (\tau_1^\bullet, \dots, \tau_m^\bullet)|\pi^\bullet$. By soundness of type inference we have that $\mathcal{A}\pi^\bullet \vdash^m (\varphi(R_1), \dots, \varphi(R_m)) : (\tau_1^\bullet, \dots, \tau_m^\bullet)$. For each rule $(f\ \overline{t_n} \rightarrow e) \in \mathcal{P}$ there will be a sub-derivation $\mathcal{A}\pi^\bullet \vdash^m \mathit{pair}\ (\lambda\overline{t_n}.e)\ f : \tau_i^\bullet$, so $\mathcal{A}\pi^\bullet \vdash^m \lambda\overline{t_n}.e : \tau_i^\bullet$. As $\mathcal{A}$ contains type variables as assumptions for the function symbols in the program, $\mathcal{A}\pi^\bullet(f) = \tau_i^\bullet$. By Lemma 18 we know that $\mathcal{A}\pi' \vdash^m \lambda\overline{t_n}.e : \tau_i'^\bullet$ such that $\mathit{flat}(\tau_i'^\bullet) = \mathit{flat}(\tau_i^\bullet)$. The rule $f\ \overline{t_n} \rightarrow e$ is then well-typed w.r.t. $\mathcal{A}\pi'$ because $\mathcal{A}\pi' \vdash^m \lambda\overline{t_n}.e : \tau_i'^\bullet$ and $\mathit{flat}(\tau_i'^\bullet) = \mathit{flat}(\tau_i^\bullet)$ is a variant of $\mathcal{A}\pi'(f) = \mathit{flat}(\tau_i^\bullet)$—notice that as $\mathcal{A}$ contains type variables as assumptions for the function symbols in the program and $\mathcal{A}\pi^\bullet(f) = \tau_i^\bullet$ then $\mathcal{A}\pi'(f)$ must be $\mathit{flat}(\tau_i^\bullet)$.
Finally, the whole program is well-typed w.r.t. $\mathcal{A}\pi'$—$wt^m_{\mathcal{A}\pi'}(\mathcal{P})$—since all the rules are well-typed w.r.t. $\mathcal{A}\pi'$. □