# Well-typed Narrowing with Extra Variables in Functional-Logic Programming (Extended Version)*

Technical Report SIC-11-11 (Revised March 16, 2012)
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

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

## Abstract

Narrowing is the usual computation mechanism in functional-logic programming (FLP), where bindings for free variables are found at the same time that expressions are reduced. These free variables may be already present in the goal expression, but they can also be introduced during computations by the use of program rules with extra variables. However, it is known that narrowing in FLP generates problems from the point of view of types, problems that can only be avoided using type information at run-time. Nevertheless, most FLP systems use static typing based on Damas-Milner type system and they do not carry any type information in execution, thus ill-typed reductions may be performed in these systems. In this paper we prove, using the let-narrowing relation as the operational mechanism, that types are preserved in narrowing reductions provided the substitutions used preserve types. Based on this result, we prove that types are also preserved in narrowing reductions without type checks at run-time when higher order (HO) variable bindings are not performed and most general unifiers are used in unifications, for programs with transparent patterns. Then we characterize a restricted class of programs for which no binding of HO variables happens in reductions, identifying some problems encountered in the definition of this class. To conclude, we use the previous results to show that a simulation of needed narrowing via program transformation also preserves types.

***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, narrowing, extra variables, type systems

## 1. Introduction

**Functional-logic programming (FLP).** Functional logic languages [3, 15, 29] like Toy [23] or Curry [16] can be described as an extension of a lazy purely-functional language similar to Haskell [18], that has been enhanced with logical features, in particular logical variables and non-deterministic functions. Disregarding some syntactic conventions, the following program defining standard list concatenation is valid in all the three mentioned languages:

$$[\,] + +Ys = Ys \qquad [X \mid Xs] + +Ys = [X \mid Xs + +Ys]$$

*Logical variables* are just free variables that get bound during the computation in a way similar to what it is done in logic programming languages like Prolog [11]. This way FLP shares with logic programming the ability of computing with partially unkown data. For instance, assuming a suitable definition and implementation of equality $==$, the following is a natural FLP definition of a predicate (a *true*-valued function) *sublist* stating that a given list $Xs$ is a sublist of $Ys$:

$$\begin{array}{l} sublist\ Xs\ Ys = cond\ (Us + +Xs + +Vs == Ys)\ true \\ cond\ true\ X = X \end{array}$$

Notice that the rule for *sublist* is not valid in a functional language due to the presence of the variables $Us$ and $Vs$, which do not occur in the left hand side of the program rule. They are called *extra variables*. Using *cond* and extra variables makes easy translating pure logic programs into functional logic ones[1]. For instance, the logic program using Peano's natural numbers $z$ (zero) and $s$ (successor)

$$\begin{array}{l} add(z, X, X). \\ add(s(X), Y, s(Z)) :- add(X, Y, Z). \\ even(X) :- add(Y, Y, X). \end{array}$$

can be transformed into the following functional logic one:

$$\begin{array}{l} add\ z\ X\ Y = cond\ (X == Y)\ true \\ add\ (s\ X)\ Y\ (s\ Z) = add\ X\ Y\ Z \\ even\ X = add\ Y\ Y\ X \end{array}$$

Notice that the rule for *even* is another example of FLP rule with an extra variable $Y$. The previous examples show that, contrary to

* This paper is the extended version of "**Well-typed Narrowing with Extra Variables in Functional-Logic Programming**", appeared in *Proceeding of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation (PEPM'12)*, pages 83–92, ACM.

[1] As a secondary question here, notice that using *cond* is needed if $==$, as usual, is a two-valued function returning *true* or *false*. Defining directly $sublist\ Xs\ Ys = (Us + +Xs + +Vs == Ys)$ would compute wrong answers: evaluating $sublist$ [1] $[1, 2]$ produces $true$ but also the wrong value *false*, because there are values of the extra variables $Us$ and $Vs$ such that $Us + +[1] + +Vs == [1, 2]$ evaluates to *false*.

the usual practice in functional programming, free variables may appear freely during the computation, even when starting from an expression without free variables. Nevertheless, despite these connections with logic programming, owing to the functional characteristics of FLP languages, like the nesting of function applications instead of SLD resolution, several variants and formulations of narrowing [19] have been adopted as the computation mechanism in FLP. There are several operational semantics for computing with logical and extra variables [15, 24, 29], and this kind of variables are supported in every modern FLP system.

As FLP languages were already non-deterministic due to the different possible instantiations of logical variables—these are handled by means of a backtracking mechanism similar to that of Prolog—it was natural that these languages eventually evolved to include so-called *non-deterministic functions*, which are functions that may return more than one result for the same input. These functions are expressed by means of program rules whose left hand sides overlap, and that are tried in order by backtracking during the computation, instead of taking a first fit or best fit approach like in pure functional languages. The combination of lazy evaluation and non-deterministic functions gives rise to several semantic options, being *call-time choice* semantics [13] the option adopted by the majority of modern FLP implementations. This point can be easily understood by means of the following program example:

$$coin \to z \quad coin \to s\ z \quad dup\ X \to (X, X)$$

In this example *coin* is a non-deterministic expression, as it can be reduced both to the values *z* and *s z*. But the point is that, according to call-time choice the expression *dup coin* evaluates to *(z, z)* and *(s z, s z)* but not to *(z, s z)* nor *(s z, z)*. Operationally, call-time choice means that all copies of a non-deterministic subexpression, like *coin* in the example, created during the computation reduction share the same value. In Section 2.2 we will see a simple formulation of narrowing for programs with extra variables, that also respects call-time choice, which will be used as the operational procedure for this paper.

Apart from these features, in the Toy system left hand sides of program rules can use not only first order patterns like those available in Haskell programs, but also higher order patterns (*HO-patterns*), which essentially are partial applications of function or constructor symbols to other patterns. This corresponds to an *intensional* view of functions, i.e., different descriptions of the same 'extensional' function can be distinguished by the semantics, and it is formalized and semantically characterized with detail in the HO-CRWL[2] logic for FLP [12]. This is not an exoticism: it is known [24] that extensionality is not a valid principle within the combination of higher order functions, non-determinism and call-time choice. HO-patterns are a great expressive feature [29], however they may have some bad interferences with types, as we will see later in the paper.

Because of all the presented features, FLP languages can be employed to write concise and expressive programs, specially for search problems, as it was explored in [3, 15, 29].

**FLP and types.** Current FLP languages are strongly typed. Apart from programming purposes, types play a key role in some program analysis or transformations for FLP, as detecting deterministic computations [17], translation of higher order into first order programs [4], or transformation into Haskell [8]. From the point of view of types FLP has not evolved much from Damas-Milner type system [9], so current FLP systems use an almost direct adaptation of that classic type system. However, that approach lacks type preservation during evaluation, even for the restricted case where we drop logical and extra variables. It is known from afar [14] that, even in that simplified scenario, HO-patterns break the type preservation property. In particular that allows us to create polymorphic casting functions [7]—functions with type $\forall\alpha, \beta.\alpha \to \beta$, but that behave like the identity wrt. the reduction of expressions. This has motivated the development of some recent works dealing with *opaque HO-patterns* [22], or liberal type systems for FLP [21]. There are also some preliminary works concerning the incorporation of *type classes* to FLP languages [25, 28], but this feature is still in an experimental phase in current systems.

Regardless of the expressiveness of extra variables these are usually out the scope of the works dealing with types and FLP, in particular in all the aforementioned. But these variables are a distinctive feature of FLP systems, hence in this work our *main goal* is to investigate the properties of a variation of the Damas-Milner type system that is able to handle extra variables, giving an abstract characterization of the problematic issues—most of them were already identified in the seminal work [14]—and then determining sufficient conditions under which type preservation is recovered for programs with extra variables evaluated with narrowing. In particular we are interested in preserving types without having to use type information at run-time, in contrast to what it is done in previous proposals [14].

The rest of the paper is organized as follows. Section 2 contains some technical preliminaries and notations about programs and expressions, and the formulation of the let-narrowing relation $\rightsquigarrow^l$, which will be used as the operational mechanism for this paper. In Section 3 we present our type system and study those interactions with let-narrowing that lead to the loss of type preservation. Then we define the well-typed let-narrowing relation $\rightsquigarrow^{lwt}$, a restriction of $\rightsquigarrow^l$ that preserves types relying on the abstract notion of well-typed substitution. To conclude that section we present $\rightsquigarrow^{lmgu}$, another restriction of $\rightsquigarrow^l$ that is able to preserve types without using type information—in contrast to $\rightsquigarrow^{lwt}$, which uses types at each step to determine that the narrowing substitution is well-typed—at the price of losing some completeness. To cope with this lack of completeness, in Section 4 we look for sufficient conditions under which the narrowing relation $\rightsquigarrow^{lmgu}$ is complete wrt. the computation of well-typed solutions, thus identifying a class of programs for which completeness is recovered, and whose expressiveness is then investigated. In Section 5 we propose a simulation of needed narrowing with $\rightsquigarrow^{lmgu}$ via two well-known program transformations, and show that it also preserves types. The class of programs supported in that section is specially relevant, as it corresponds to a simplified version of the Curry language. Finally Section 6 summarizes some conclusions and future work. Fully detailed proofs, including some auxiliary results, can be found in Appendix A.

## 2. Preliminaries

### 2.1 Expressions and programs

We consider a set of *functions* symbols $f, g, \ldots \in FS$ and *constructor* symbols $c, d, \ldots \in CS$, each $h \in FS \cup CS$ with an associated arity $ar(h)$. We also consider a denumerable set of *data variables* $X, Y, \ldots \in \mathcal{V}$. The notation $\overline{o_n}$ stands for a sequence $o_1, \ldots, o_n$ of $n$ syntactic elements $o$, being $o_i$ the $i^{th}$ element. Figure 1 shows the syntax of patterns $t \in Pat$ and expressions $e \in Exp$. 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* $HOPat = Pat \setminus FOPat$, i.e., patterns containing some partial application of a symbol of the signature. Expressions $X\ \overline{e_n}$ are called *variable application* when $n > 0$, and expressions with the form $h\ \overline{e_n}$ are called *junk* if $h \in CS$ and $n > ar(h)$ or *active* if $h \in FS$ and $n \geq ar(h)$. The set of *free* and *bound* variables of an expression $e$—$fv(e)$ and $bv(e)$ resp.—are defined

[2] CRWL [13] stands for *Constructor Based Rewriting Logic*; HO-CRWL is a higher order extension of it.

| | | | |
|---|---|---|---|
| Data variable | | | $X$,$Y$ ... |
| Function symbol | | | $f$,$g$ ... |
| Constructor symbol | | | $c$,$d$ ... |
| Non-variable symbol | $h$ | ::= | $c \mid f$ |
| Symbol | $s$ | ::= | $X \mid c \mid f$ |
| *Pat* | $t, p$ | ::= | $X$ |
| | | | $\mid c\ \overline{t_n}$ if n $\leq$ ar(c) |
| | | | $\mid f\ \overline{t_n}$ if n $<$ ar(f) |
| *FOPat* | $fot$ | ::= | $X \mid c\ \overline{fot_n}$ if n = ar(c) |
| *Exp* | $e, r$ | ::= | $X \mid c \mid f \mid e_1\ e_2$ |
| | | | $\mid$ *let* $X = e_1$ *in* $e_2$ |
| $PSubst$ | $\theta$ | ::= | $[\overline{X_n \mapsto t_n}]$ |
| $Cntxt$ | $\mathcal{C}$ | ::= | $[\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C}$ |
| | | | $\mid$ *let* $X = \mathcal{C}$ *in* $e$ |
| | | | $\mid$ *let* $X = e$ *in* $\mathcal{C}$ |
| Program rule | $R$ | ::= | $f\ \overline{t_n} \rightarrow e$ if ar(f) = n |
| Program | $\mathcal{P}$ | ::= | $\{\overline{R_n}\}$ |
| Type variable | | | $\alpha$,$\beta$ ... |
| Type constructor | | | $C$ |
| Simple type | $\tau$ | ::= | $\alpha \mid \tau_1 \rightarrow \tau_2$ |
| | | | $\mid C\ \overline{\tau_n}$ if n = ar(C) |
| Type-scheme | $\sigma$ | ::= | $\forall\overline{\alpha_n}.\tau$ |
| Set of assumptions | $\mathcal{A}$ | ::= | $\{\overline{s_n : \sigma_n}\}$ |
| $TSubst$ | $\pi$ | ::= | $[\overline{\alpha_n \mapsto \tau_n}]$ |

**Figure 1.** Syntax of programs and types

in the usual way. Notice that let-expressions are not recursive, so $fv(let\ X = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus \{X\})$. The set $var(e)$ is the set containing all the variables in $e$, both free and bound. Notice that for patterns $var(t) = fv(t)$.

*Contexts* $\mathcal{C} \in Cntxt$ are expressions with one hole, and the application of $\mathcal{C}$ to $e$—written $\mathcal{C}[e]$—is the standard. The notion of free and bound variables are extended in the natural way to contexts: $fv(\mathcal{C}) = fv(\mathcal{C}[h])$ for any $h \in FS \cup CS$ with $ar(h) = 0$, and $bv(\mathcal{C})$ is defined as $bv([\,]) = \emptyset$, $bv(\mathcal{C}\ e) = bv(\mathcal{C})$, $bv(e\ \mathcal{C}) = bv(\mathcal{C})$, $bv(let\ X = \mathcal{C}\ in\ e) = bv(\mathcal{C})$, $bv(let\ X = e\ in\ \mathcal{C}) = \{X\} \cup bv(\mathcal{C})$.

*Data substitution* $\theta \in PSubst$ are finite maps from data variables to patterns $[\overline{X_n \mapsto t_n}]$. We write $\epsilon$ for the empty substitution, $dom(\theta)$ for the domain of $\theta$ and $vran(\theta) = \bigcup_{X \in dom(\theta)} fv(X\theta)$. Given $A \subseteq \mathcal{V}$, the notation $\theta|_A$ represents the restriction of $\theta$ to $D$, and $\theta|_{\setminus A}$ is a shortcut for $\theta|_{\mathcal{V} \setminus A}$. Substitution application over data variables and expressions is defined in the usual way.

Program rules $R$ have the form $f\ \overline{t_n} \rightarrow e$, where $ar(f) = n$ and $\overline{t_n}$ is *linear*, i.e., there is no repetition of variables. Notice that we allow extra variables, so it could be the case that $e$ contains variables which do not appear in $\overline{t_n}$. A program $\mathcal{P}$ is a set of program rules.

### 2.2 Let-narrowing

Let-narrowing [24] is a narrowing relation devised to effectively deal with logical and extra variables, that is also sound and complete wrt. HO-CRWL [12], a standard logic for higher order FLP with call-time choice. Figure 2 contains the rules of the let-narrowing relation $\rightsquigarrow^l$. 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, but keeping the implied term graph untouched. The (Narr) rule performs function application, finding the bindings for the free variables needed to be able to apply the rule, and possibly introducing new variables if the program rule contains some extra variables. Notice that it does not require the use of a most general unifier (mgu) so any unifier can be used. As we will see in Section 3, this later point should be refined in order to ensure type preservation. Rules (VAct) and (VBind) produce HO bindings for variable applications, and are needed for let-narrowing to be complete. These rules are particularly problematic because they have to generate speculative bindings that may involve any function of the program, contrary to (Narr) where the computation of bindings is directed by the program rules for $f$. Later on we will see how this "wild" nature of the bindings generated by these rules poses especially hard problems to type preservation. Finally, (Contx) allows to apply a narrowing rule in any part of the expression, protecting bound variables from narrowing and avoiding variable capture.

## 3. Type Preservation

In this section we first present the type system we will use in this work, which is a simple variation of Damas-Milner typing enhanced with support for extra variables. Then we show some examples of $\rightsquigarrow^l$-reductions not preserving types (Section 3.2). Based on the ideas that emerge from these examples, in Section 3.3 we develop a new let-narrowing relation $\rightsquigarrow^{lwt}$ that preserves types. This new relation uses only *well-typed substitutions* in each step, which gives an abstract and general characterization of the requirements a narrowing relation must fulfil in order to preserve types, but it still needs to perform type checks at run-time. To solve this problem, in Section 3.4 we present a restricted let-narrowing $\rightsquigarrow^{lmgu}$ which only uses mgu's as unifiers and drops the problematic rules (VAct) and (VBind). The main advantage of this relation is that if the patterns that can appear in program rules are limited then mgu's are always well-typed, thus obtaining type preservation without using type information at run-time. Sadly this comes at a price, as $\rightsquigarrow^{lmgu}$ loses some completeness wrt. HO-CRWL.

### 3.1 A type system for extra variables

In Figure 1 we can find the usual syntax for *simple types* $\tau$ and *type-schemes* $\sigma$. For a simple type $\tau$, the set of *free type variables*—denoted $ftv(\tau)$—is $var(\tau)$, and for type-schemes $ftv(\forall\overline{\alpha_n}.\tau) = var(\tau) \setminus \{\overline{\alpha_n}\}$. A type-scheme is *closed* if $ftv(\sigma) = \emptyset$. We say that a type-scheme is *k-transparent* if it can be written as $\forall\overline{\alpha_n}.\overline{\tau_k} \rightarrow \tau$ such that $var(\overline{\tau_k}) \subseteq var(\tau)$.

A set of assumptions $\mathcal{A}$ is a set of the form $\{\overline{s_n : \sigma_n}\}$ such that the assumption for variables are simple types. If $(s_i : \sigma_i) \in \mathcal{A}$ we write $\mathcal{A}(s_i) = \sigma_i$. For sets of assumptions we define $ftv(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. The union of set of assumptions is denoted by $\oplus$ with the usual meaning: $\mathcal{A} \oplus \mathcal{A}'$ contains all the assumptions in $\mathcal{A}'$ as well as the assumptions in $\mathcal{A}$ for those symbols not appearing in $\mathcal{A}'$. Based on the previous notion of k-transparency, we say a pattern $t$ is *transparent* wrt. $\mathcal{A}$ if $t \in \mathcal{V}$ or $t \equiv h\ \overline{t_n}$ where $\mathcal{A}(h)$ is n-transparent and $\overline{t_n}$ are transparent patterns. We also say a constructor symbol $c$ is transparent wrt. $\mathcal{A}$ if $\mathcal{A}(c)$ is n-transparent, where $ar(c) = n$.

*Type substitutions* $\pi \in TSubst$ are mappings from type variables to simple types, where $dom$ and $vran$ are defined similarly to data substitutions. Application of type substitutions to simple types is defined in the natural way, and for type-schemes consists in applying the substitution only to their free variables. This notion is extended to set of assumptions: $\{\overline{s_n : \sigma_n}\}\pi = \{\overline{s_n : \sigma_n\pi}\}$. We say $\tau$ is a *generic instance* of $\sigma \equiv \forall\overline{\alpha_n}.\tau'$ if $\tau = \tau'[\overline{\alpha_n \mapsto \tau_n}]$ for some $\overline{\tau_n}$, written $\sigma \succ \tau$. Finally, $\tau$ is a *variant* of $\sigma \equiv \forall\overline{\alpha_n}.\tau'$ (denoted by $\sigma \succ_{var} \tau$) if $\tau = \tau'[\overline{\alpha_n \mapsto \beta_n}]$ where $\overline{\beta_n}$ are fresh type variables.

Figure 3 contains the typing rules for expressions considered in this work, which constitute a variation of Damas-Milner typing

**(LetIn)** $e_1\ e_2 \rightsquigarrow^l_\epsilon\ let\ X = e_2\ in\ e_1\ X$,    if $e_2$ is an active expression, variable application, junk or let-rooted expression, for $X$ fresh.
**(Bind)** $let\ X = t\ in\ e\ \rightsquigarrow^l_\epsilon\ e[X \mapsto t]$,    if $t \in Pat$
**(Elim)** $let\ X = e_1\ in\ e_2 \rightsquigarrow^l_\epsilon e_2$,    if $X \notin fv(e_2)$
**(Flat)** $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3\ \rightsquigarrow^l_\epsilon\ let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$,    if $Y \notin fv(e_3)$
**(LetAp)** $(let\ X = e_1\ in\ e_2)\ e_3 \rightsquigarrow^l_\epsilon\ let\ X = e_1\ in\ e_2\ e_3$,   if $X \notin fv(e_3)$
**(Narr)** $f\ \overline{t_n} \rightsquigarrow^l_\theta r\theta$,    for any fresh variant $(f\ \overline{p_n} \to r) \in \mathcal{P}$ and $\theta$ such that $f\ \overline{t_n}\theta \equiv f\ \overline{p_n}\theta$.
**(VAct)** $X\ \overline{t_k} \rightsquigarrow^l_\theta r\theta$,    if $k > 0$, for any fresh variant $(f\ \overline{p} \to r) \in \mathcal{P}$ and $\theta$ such that $(X\ \overline{t_k})\theta \equiv f\ \overline{p}\theta$
**(VBind)** $let\ X = e_1\ in\ e_2 \rightsquigarrow^l_\theta e_2\theta[X \mapsto e_1\theta]$,   if $e_1 \notin Pat$, for any $\theta$ that makes $e_1\theta \in Pat$, provided that $X \notin (dom(\theta) \cap vran(\theta))$
**(Contx)** $\mathcal{C}[e] \rightsquigarrow^l_\theta \mathcal{C}\theta[e']$,    for $\mathcal{C} \neq [\,]$, $e \rightsquigarrow^l_\theta e'$ using any of the previous rules, and:
*i)* $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$
*ii)* • if the step is (Narr) or (VAct) using $(f\ \overline{p_n} \to r) \in \mathcal{P}$ then $vran(\theta|_{\setminus var(\overline{p_n})}) \cap bv(\mathcal{C}) = \emptyset$
• if the step is (VBind) then $vran(\theta) \cap bv(\mathcal{C}) = \emptyset$.

**Figure 2.** Let-narrowing relation $\rightsquigarrow^l$

that now is able to handle extra variables. The main novelty wrt. a regular formulation of Damas-Milner typing with support for pattern matching is that now the $(\Lambda)$ rule considers extra variables in $\lambda$-abstractions: in addition to guessing types for the variables in the pattern $t$, it also guesses types for the free variables of $\lambda t.e$, which correspond to extra variables. Although $\lambda$-abstractions are expressions not included in the syntax of programs showed in Figure 1 and thus they cannot appear in the expressions to reduce[3], we use them as the basis for the notions of well-typed rule and program. Essentially, for each program rule we construct an associated $\lambda$-abstraction so the rule is well-typed iff the corresponding $\lambda$-abstraction is well-typed. This is reflected in the following definition of *program well-typedness*, an important property assuring that assumptions over functions are related to their rules:

DEFINITION 3.1 (Well-typed program wrt. $\mathcal{A}$). *A program rule $f \to e$ is well-typed wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ where $\mathcal{A}(f) \succ_{var} \tau$, $\{\overline{X_n}\} = fv(e)$ and $\overline{\tau_n}$ are some simple types. A program rule $(f\ \overline{p_n} \to e)$ (with $n > 0$) is well-typed wrt. $\mathcal{A}$ iff $\mathcal{A} \vdash \lambda p_1 \ldots \lambda p_n.e : \tau$ with $\mathcal{A}(f) \succ_{var} \tau$. A program $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$ if all its rules are well-typed wrt. $\mathcal{A}$.*

This definition is the same as the one from [22] but it has a different meaning, as it is based on a different definition for the $(\Lambda)$ rule. Notice that the case $f \to e$ must be handled independently because it does not have any argument. In this case the $(\Lambda)$ rule is not used to derive the type for $e$, so the types for the extra variables would not be guessed.

An expression $e$ is well-typed wrt. $\mathcal{A}$ iff $\mathcal{A} \vdash e : \tau$ for some type $\tau$, written as $wt_{\mathcal{A}}(e)$. We will use the metavariable $\mathcal{D}$ to denote particular type derivations $\mathcal{A} \vdash e : \tau$. If $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$ we write $wt_{\mathcal{A}}(\mathcal{P})$.

### 3.2 Let-narrowing does not preserve types

Now we will see how let-narrowing interacts with types. It is easy to see that let-narrowing steps $\rightsquigarrow^l$ which do not generate bindings for the logical variables—i.e., those using the rules (LetIn), (Bind), (Elim), (Flat) and (LetAp)—preserve types trivially. This is not very surprising because, as we showed in Section 2.2, those steps just change the textual representation of the implied term graph. However, steps generating non trivial bindings can break type preservation easily:

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

$$\text{(APP)}\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

$$(\Lambda)\ \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \qquad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau}\ \text{if } \{\overline{X_n}\} = var(t) \cup fv(\lambda t.e)$$

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

**Figure 3.** Type System

EXAMPLE 3.2. *Consider the function and defined by the rules $\{and\ true\ X \to X, and\ false\ X \to false\}$ with type $(bool \to bool \to bool)$ and the constructor symbols for Peano's natural numbers z and s, with types $(nat)$ and $(nat \to nat)$ respectively. Starting from the expression and true Y—which has type bool when Y has type bool—we can perform the let-narrowing step:*

$$and\ true\ Y \rightsquigarrow^l_{[X_1 \mapsto z, Y \mapsto z]} z$$

*This (Narr) step uses the fresh program rule $(and\ true\ X_1 \to X_1)$, but the resulting expression z does not have type bool.*

*The cause of the loss of type preservation is that the unifier $\theta_1 = [X_1 \mapsto z, Y \mapsto z]$ used in the (Narr) step is ill-typed, because it replaces the boolean variables $X_1$ and $Y$ by the natural z. The problem with $\theta_1$ is that it instantiates the variables too much, and without using any criterion that ensures that the types of the expressions in its range are adequate.*

*We have just seen that using the (Narr) rule with an ill-typed unifier may lead to breaking type preservation because of the instantiation of logical variables, like the variable Y above. We may reproduce the same problem easily with extra variables, just consider the function f with type bool defined by the rule $(f \to and\ true\ X)$ for which we can perform the following let-narrowing step:*

$$f \rightsquigarrow^l_{[X_2 \mapsto z]} and\ true\ z$$

[3] As there is no general consensus about the semantics of $\lambda$-abstractions in the FLP community, due to their interactions with non-determinism and logical variables, we have decided to leave $\lambda$-abstractions out of programs and evaluating expressions, thus following the usual applicative programming style of the HO-CRWL logic.

*using (Narr) with the fresh rule $(f \to and\ true\ X_2)$. The resulting expression is obviously ill-typed, and so type preservation is broken again because the substitution used in (Narr) instantiates variables too much and without assuring that the expression in its range have the correct types. The interested reader may easily check that this is also a valid let-rewriting step [24], thus showing that extra variables break type preservation even in the restricted scenario where we drop logical variables. Hence, the type systems in the papers mentioned at the end of Section 1 lose type preservation if we allow extra variables in the programs.*

However, the (Narr) rule is not the only one which can break type preservation. The rules (VAct) and (VBind) also lead to problematic situations:

EXAMPLE 3.3. *Consider the functions and symbols from Example 3.2. Using the rule (VAct) it is possible to perform the step*

$$s\ (F\ z) \rightsquigarrow^{l}_{[F \mapsto and\ false, X_3 \mapsto z]} s\ false$$

*with the fresh rule $(and\ false\ X_3 \to false)$. Clearly $s\ (F\ z)$ has type $nat$ and $F$ has type $(nat \to nat)$, but the resulting expression is ill-typed. As before, the reason is an ill-typed binding for $F$, which binds $F$ with a pattern of type $(bool \to bool)$.*

*On the other hand, we can perform the step*

$$let\ X = F\ z\ in\ s\ X \rightsquigarrow^{l}_{[F \mapsto and]} s\ (and\ z)$$

*using the rule (VBind). The expression $let\ X = F\ z\ in\ s\ X$ has type $nat$ when $F$ has type $(nat \to nat)$, but the resulting expression is ill-typed. The cause of the loss of type preservation is again an ill-typed substitution binding, in this case the one for $F$ which assigns a pattern of type $(bool \to bool \to bool)$ to a variable of type $(nat \to nat)$.*

Notice that ill-typed substitutions do not break type preservation necessarily. For example the step $and\ false\ X \rightsquigarrow^{l}_{\theta_5} false$ using (Narr) with the fresh rule $(and\ false\ X_5 \to false)$ preserves types, although it can use the ill-typed unifier $\theta_5 \equiv [X \mapsto z, X_5 \mapsto z]$. However, avoiding ill-typed substitutions is a sufficient condition which guarantees type preservation, as we will see soon. Besides, it is important to remark that the bindings for the free variables of the starting expression that are computed in a narrowing derivation are as important as the final value reached at the end of the derivation, because these bindings constitute a solution for the starting expression if we consider it as a goal to be solved, just like the goal expressions used in logic programming. That allows us to use predicate functions like the function $sublists$ in Section 1 with some variables as their arguments, i.e., using some arguments in Prolog-like output mode. Therefore, well-typedness of the substitutions computed in narrowing reductions is also important and the restriction to well-typed substitutions is not only reasonable but also desirable, as it ensures that the solutions computed by narrowing respect types.

### 3.3 Well-typed let-narrowing $\rightsquigarrow^{lwt}$

In this section we present a narrowing relation $\rightsquigarrow^{lwt}$ which is smaller than $\rightsquigarrow^{l}$ in Figure 2 but that preserves types. The idea behind $\rightsquigarrow^{lwt}$ is that it only considers steps $e \rightsquigarrow^{l}_{\theta} e'$ using well-typed programs where the substitution $\theta$ is also well-typed. We say a substitution is well-typed when it replaces data variables by patterns of the same type. Formally:

DEFINITION 3.4 (Well-typed substitution). *A data substitution $\theta$ is well-typed wrt. $\mathcal{A}$, written $wt_{\mathcal{A}}(\theta)$, if $\mathcal{A} \vdash X\theta : \mathcal{A}(X)$ for every $X \in dom(\theta)$.*

Notice that according to the definition of set of assumptions, $\mathcal{A}(X)$ is always a simple type.

As it is usual in narrowing relations, let-narrowing steps can introduce new variables that do not occur in the original expression. Moreover, this new variables do not come only from extra variables but from fresh variants of program rules—using (Narr) and (VAct)—or from invented patterns—using (VBind). Therefore, we need to consider some suitable assumptions over these new variables. However, that set of assumptions over the new variables is not arbitrary but it is closely related to the step used:

EXAMPLE 3.5 ($\mathcal{A}$ associated to a (Narr) step). *Consider the function $f$ with type $\forall\alpha.\alpha \to [\alpha]$ defined with the rule $f\ X \to [X, Y]$. We can perform the narrowing step $f\ true \rightsquigarrow^{l}_{\theta} [true, Y_1]$ using (Narr) with the fresh variant $f\ X_1 \to [X_1, Y_1]$ and $\theta \equiv [X_1 \mapsto true]$. Since the original expression is $f\ true$, it is clear that $X_1$ must have type $bool$ in the new set of assumptions. Moreover, $Y_1$ must have the same type since it appears in a list with $X_1$. Therefore in this concrete step the associated set of assumptions is $\{X_1 : bool, Y_1 : bool\}$.*

The following definition establishes when a set of assumptions is associated to a step. Notice that due to the particularities of the rules (VAct) and (VBind), in some cases there is not such set or there are several associated sets.

DEFINITION 3.6 ($\mathcal{A}$ associated to $\rightsquigarrow^{l}$ steps). *Given a type derivation $\mathcal{D}$ for $\mathcal{A} \vdash e : \tau$ and $wt_{\mathcal{A}}(\mathcal{P})$, a set of assumptions $\mathcal{A}'$ is associated to the step $e \rightsquigarrow^{l}_{\theta} e'$ iff:*

- *$\mathcal{A}' \equiv \emptyset$ and the step is (LetIn), (Bind), (Elim), (Flat) or (LetAp).*
- *If the step is (Narr) then $f\ \overline{t_n} \rightsquigarrow^{l}_{\theta} r\theta$ using a fresh variant $(f\ \overline{p_n} \to r) \in \mathcal{P}$ and substitution $\theta$ such that $(f\ \overline{p_n})\theta \equiv (f\ \overline{t_n})\theta$. Since $\mathcal{D}$ is a type derivation for $\mathcal{A} \vdash f\ \overline{t_n} : \tau$, it will contain a derivation $\mathcal{A} \vdash f : \overline{\tau_n} \to \tau$. The rule $f\ \overline{p_n} \to r$ is well-typed by $wt_{\mathcal{A}}(\mathcal{P})$, so we also have (when the rule is $f \to e$ it is similar):*

$$(\Lambda)\,\dfrac{\mathcal{A} \oplus \mathcal{A}_1 \vdash p_1 : \tau'_1 \qquad (\Lambda)\,\dfrac{\begin{array}{c}\mathcal{A} \oplus \mathcal{A}_1 \ldots \oplus \mathcal{A}_n \vdash p_n : \tau'_n \\ \mathcal{A} \oplus \mathcal{A}_1 \ldots \oplus \mathcal{A}_n \vdash r : \tau'\end{array}}{\vdots}}{\mathcal{A} \vdash \lambda p_1 \ldots \lambda p_n.r : \overline{\tau'_n} \to \tau'}$$

  *where $\overline{\mathcal{A}_n}$ are the set of assumptions over variables introduced by $(\Lambda)$ and $\overline{\tau'_n} \to \tau'$ is a variant of $\mathcal{A}(f)$. Therefore $(\overline{\tau'_n} \to \tau')\pi \equiv \overline{\tau_n} \to \tau$ for some type substitution $\pi$ whose domain are fresh type variables from the variant. In this case $\mathcal{A}'$ is associated to the (Narr) step if $\mathcal{A}' \equiv (\mathcal{A}_1 \oplus \ldots \oplus \mathcal{A}_n)\pi$.*
- *If the step is (VAct) then we have $X\ \overline{t_k} \rightsquigarrow^{l}_{\theta} r\theta$ for a fresh variant $(f\ \overline{p_n} \to r) \in \mathcal{P}$ and substitution $\theta$ such that $(X\ \overline{t_k})\theta \equiv f\ \overline{p_n}\theta$. Since $\mathcal{D}$ is a type derivation for $\mathcal{A} \vdash X\ \overline{t_k} : \tau$, it will contain a derivation $\mathcal{A} \vdash X : \overline{\tau_k} \to \tau$. The rule $f\ \overline{p_n} \to r$ is well-typed by $wt_{\mathcal{A}}(\mathcal{P})$, so we have a type derivation $\mathcal{A} \vdash \lambda p_1 \ldots \lambda p_n.r : \overline{\tau'_n} \to \tau'$ as in the (Narr) case (similarly when the rule is $f \to e$). Let $\overline{\tau''_k}$ be $\tau'_{n-k+1} \to \tau'_{n-k+2} \cdots \to \tau'_n$, i.e., the last $k$ types in $\overline{\tau'_n}$. If $\mathcal{A}' \equiv (\mathcal{A}_1 \oplus \ldots \oplus \mathcal{A}_n)\pi$ for some substitution $\pi$ such that $(\overline{\tau''_k} \to \tau')\pi \equiv \overline{\tau_k} \to \tau$ and $fv(\mathcal{A}) \cap dom(\pi) = \emptyset$, then $\mathcal{A}'$ is associated to the (VAct) step.*
- *Any $\mathcal{A}' \equiv \{\overline{X_n : \tau_n}\}$ is associated to a (VBind) step, if $\overline{X_n}$ are those data variables introduced by $vran(\theta)$—they do not appear in $\mathcal{A}$—and $\overline{\tau_n}$ are simple types.*
- *$\mathcal{A}'$ is associated to a (Contx) step if it is associated to its inner step.*

*A set of assumptions $\mathcal{A}'$ is associated to $n \rightsquigarrow^{l}$ steps $(e_1 \rightsquigarrow^{l} e_2 \ldots \rightsquigarrow^{l} e_{n+1})$ if $\mathcal{A}' \equiv \mathcal{A}'_1 \oplus \mathcal{A}'_2 \ldots \oplus \mathcal{A}'_n$, where $\mathcal{A}'_i$ is associated to the step $e_i \rightsquigarrow^{l} e_{i+1}$ and the type derivation $\mathcal{D}_i$ for $e_i$ using $\mathcal{A} \oplus \mathcal{A}'_1 \ldots \oplus \mathcal{A}'_{i-1}$ ($\mathcal{A}' \equiv \emptyset$ if $n = 0$).*

Based on the previously introduced notions we can define a restriction of let-narrowing that only employs well-typed substitutions, that we will denote by $\rightsquigarrow^{lwt}$:

DEFINITION 3.7 ($\rightsquigarrow^{lwt}$ let-narrowing). *Consider an expression $e$, a program $\mathcal{P}$ and set of assumptions $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$ with a derivation $\mathcal{D}$ and $wt_{\mathcal{A}}(\mathcal{P})$. Then $e \rightsquigarrow^{lwt}_{\theta} e'$ iff $e \rightsquigarrow^{l}_{\theta} e'$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to $e \rightsquigarrow^{l}_{\theta} e'$, $\mathcal{D}$.*

The premises $wt_{\mathcal{A}}(e)$ and $wt_{\mathcal{A}}(\mathcal{P})$ are essential, since the associated set of assumptions wrt. $e \rightsquigarrow^{l}_{\theta} e'$ is only well defined in those cases. Note that the step $\rightsquigarrow^{lwt}$ cannot be performed if no set of associated assumptions $\mathcal{A}'$ exists. Although $\rightsquigarrow^{lwt}$ is strictly smaller than $\rightsquigarrow^{l}$—the steps in Examples 3.2 and 3.3 are not valid $\rightsquigarrow^{lwt}$-steps—it enjoys the intended type preservation property:

THEOREM 3.8 (Type preservation of $\rightsquigarrow^{lwt}$). *If $wt_{\mathcal{A}}(\mathcal{P})$, $e \rightsquigarrow^{lwt^*}_{\theta} e'$ and $\mathcal{A} \vdash e : \tau$ then $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

The previous result is the main contribution of this paper. It states clearly that, provided that the substitutions used are well-typed, let-narrowing steps preserve types. Moreover, type preservation is guaranteed for general programs, i.e., programs containing extra variables, non-transparent constructor symbols, opaque HO-patterns ... This result is very relevant because it clearly isolates a sufficient and reasonable property that, once imposed to the unifiers, ensures type preservation. Besides, this condition is based upon the abstract notion of well-typed substitution, which is parameterized by the type system and independent of the concrete narrowing or reduction notion employed. Thus the problem of type preservation in let-narrowing reductions is clarified. New let-narrowing subrelations can be proposed for restricted classes of programs or using particular unifiers and, provided the generated substitutions are well-typed, they will preserve types. We will see an example of that in Section 3.4.

This is an important advance wrt. previous proposals like [14], where the computation of the mgu was interleaved with and inseparable from the rest of the evaluation process in the narrowing derivations. Besides, although the identification of three kinds of problematic situations for the type preservation made in that work was very valuable—especially taking into account it was one of the first studies of the subject in FLP with HO-patterns—having a more general and abstract result is also valuable for the reasons stated above.

### 3.4 Restricted narrowing using mgu's $\rightsquigarrow^{lmgu}$

The $\rightsquigarrow^{lwt}$ relation has the good property of preserving types, however it presents a drawback if used as the reduction mechanism of a FLP system: it requires the substitutions generated in each $\rightsquigarrow^{lwt}$ step to be well-typed. Since these substitutions are generated just by using the syntactic criteria expressed in the rules of the let-narrowing relation $\rightsquigarrow^{l}$, the only way to guarantee this is to perform type checks at run-time, discarding ill-typed substitutions. But, as we mentioned in Section 1, we are interested in preserving types without having to use type information at run-time. Hence, in this section we propose a new let-narrowing relation $\rightsquigarrow^{lmgu}$ which preserves types without need of type checks at run-time. The let-narrowing relation $\rightsquigarrow^{lmgu}$ is defined as:

DEFINITION 3.9 (Restricted narrowing $\rightsquigarrow^{lmgu}$). *$e \rightsquigarrow^{lmgu}_{\theta} e'$ iff $e \rightsquigarrow^{l}_{\theta} e'$ using any rule from Figure 2 except (VAct) and (VBind), and if the step is $f\ \overline{t_n} \rightsquigarrow^{l}_{\theta} r\theta$ using (Narr) with the fresh variant $(f\ \overline{p_n} \rightarrow r)$ then $\theta = mgu(f\ \overline{t_n}, f\ \overline{p_n})$.*

As explained in Section 3.2, the rules that break type preservation are (Narr), (VAct) and (VBind). The rules (VAct) and (VBind) present harder problems to preserve types since they replace HO variables by patterns. These patterns are searched in the entire space of possible patterns, producing possible ill-typed substitutions. Since we want to avoid type checks at run-time, and we have not found any syntactic criterion to forbid the generation of ill-typed substitutions by those rules, (VAct) and (VBind) have been omitted from $\rightsquigarrow^{lmgu}$. Although this makes $\rightsquigarrow^{lmgu}$ a relation strictly smaller than $\rightsquigarrow^{lwt}$, it is still meaningful: expressions needing (VAct) or (VBind) to proceed can be considered as *frozen* until other let-narrowing step instantiates the HO variable. This is somehow similar to the operational principle of *residuation* used in some FLP languages such as Curry [15, 16]. Regarding the rule (Narr), Example 3.2 shows the cause of the break of type preservation. In that example, the unifier of $and\ true\ Y$ and $and\ true\ X_1$ is $\theta_1 = [X_1 \mapsto z, Y \mapsto z]$. Although $\theta_1$ is a valid unifier, it instantiates variables unnecessarily in an ill-typed way. In other words, it does not use just the information from the program and the expression, which are well-typed, but it "invents" the pattern $z$. We can solve this situation easily using the mgu $\theta'_1 = [X_1 \mapsto Y]$, which is well-typed, so by Theorem 3.8 we can conclude that the step preserves types.

Moreover, this solution applies to any (Narr) step (under certain conditions that will be specified later): if we chose mgu's in the (Narr) rule and both the rule and the original expression are well-typed, then the mgu's will also be well-typed. This fact is based in the following result:

LEMMA 3.10 (Mgu well-typedness). *Let $\overline{p_n}$ be fresh linear transparent patterns wrt. $\mathcal{A}$ and let $\overline{t_n}$ be any patterns such that $\mathcal{A} \vdash p_i : \tau_i$ and $\mathcal{A} \vdash t_i : \tau_i$ for some type $\tau_i$. If $\theta \equiv mgu(f\ \overline{p_n}, f\ \overline{t_n})$ then $wt_{\mathcal{A}}(\theta)$.*

The restriction to fresh linear transparent patterns $\overline{p_n}$ is essential, otherwise the mgu may not be well-typed. Consider for example the constructor $cont : \forall\alpha.\alpha \rightarrow container$ and a set of assumptions $\mathcal{A}$ containing $(X : nat)$. It is clear that $p \equiv cont\ X$ is linear but non-transparent, because $cont$ is not 1-transparent. Both $p$ and $t \equiv cont\ true$ patterns have type $container$ and $mgu(f\ p, f\ t) = [X \mapsto true] \equiv \theta$ for any function symbol $f$. However the unifier $\theta$ is ill-typed as $\mathcal{A} \nvdash X\theta : \mathcal{A}(X)$, i.e., $\mathcal{A} \nvdash true : nat$. Similarly, consider the patterns $p' \equiv (Y, Y)$ and $t' \equiv (cont\ X, cont\ true)$ and a set of assumptions $\mathcal{A}$ containing $(Y : container, X : nat)$. It is easy to see that $p'$ and $t'$ have type $(container, container)$, and $p'$ is transparent but non-linear. The mgu of $f\ p'$ and $f\ t'$ is $[Y \mapsto cont\ true, X \mapsto true]$, which is ill-typed by the same reasons as before.

Due to the previous result, type preservation is only guaranteed for $\rightsquigarrow^{lmgu}$-reductions for programs such that left-hand sides of rules contain only transparent patterns. This is not a severe limitation, as it is considered in other works [14], and as we will see in the next section.

THEOREM 3.11 (Type preservation of $\rightsquigarrow^{lmgu}$). *Let $\mathcal{P}$ be a program such that left-hand sides of rules contain only transparent patterns. If $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ and $e \rightsquigarrow^{lmgu^*}_{\theta} e'$ then $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

So finally, with $\rightsquigarrow^{lmgu}$ we have obtained a narrowing relation that is able to ensure type preservation without using any type information at run-time. However, as we mentioned before, this comes at the price of losing completeness wrt. HO-CRWL, not only because we are restricted to using mgu's—which is not a severe restriction, as we will see later—but mainly because we are not able to use the rules (VAct) and (VBind) any more, which are essential for generating binding for variable applications like those in Example 3.3. We will try to mitigate that problem in Section 4.

## 4. Reductions without Variable Applications

In this section we want to identify a class of programs in which $\rightsquigarrow^{lmgu}$ is sufficiently complete so it can perform well-typed narrowing derivations without losing well-typed solutions. As can be seen in the Lifting Lemma from [24], the restriction of the let-narrowing relation $\rightsquigarrow^{l}$ that only uses mgu's in each step is complete wrt. HO-CRWL. Therefore, we strongly believe that the restriction of $\rightsquigarrow^{lwt}$ using only mgu's is complete wrt. to the computation of well-typed solutions, although proving it is an interesting matter of future work. For this reason, in this section we are only concerned about determining under which conditions $\rightsquigarrow^{lmgu}$ is complete wrt. the restriction of $\rightsquigarrow^{lwt}$ to mgu's.

Our experience shows that although we only have to assure that neither (VAct) nor (VBind) are used, the characterization of such a family of programs is harder than expected. In Section 4.1 we show the different approaches tried, explaining their lacks, that led us to a restrictive condition—Section 4.2. This condition limits the expressiveness of the programs, hence we explore the possibilities of that class of programs in Section 4.3.

### 4.1 Naive approaches

Our first attempt follows the idea that if an expression does not contain any free HO variable (free variable with a functional type of the shape $\tau \rightarrow \tau'$) then neither (VAct) nor (VBind) can be used in a narrowing step. This result is stated in the following easy Lemma:

LEMMA 4.1 (Absence of HO variables). *Let $e$ be an expression such that $wt_{\mathcal{A}}(e)$ and for every $X_i \in fv(e)$, $\mathcal{A}(X_i)$ is not a functional type. Then no step $e \rightsquigarrow^{l}_{\theta} e'$ can use (VAct) or (VBind).*

Our belief was that if an expression does not contain free HO variables and the program does not have extra HO variables, the resulting expression after a $\rightsquigarrow^{lmgu}$ step does not have free HO variables either. This is false, as the following example shows:

EXAMPLE 4.2. *Consider a constructor symbol bfc with type $bfc : (bool \rightarrow bool) \rightarrow BoolFunctContainer$ and the function $f$ with type $f : BoolFunctContainer \rightarrow bool$ defined as $\{f\ (bfc\ F) \rightarrow F\ true\}$. We can perform the narrowing reduction*

$$f\ X \rightsquigarrow^{lmgu}_{\theta} F_1\ true$$

*where $\theta \equiv [X \mapsto bfc\ F_1] = mgu(f\ X, f\ (bfc\ F_1))$. The free variable $F_1$ introduced has a functional type, however the original expression has not any free HO variable—$X$ has the ground type $BoolFunctContainer$. Moreover, the program does not contain extra variables at all.*

The previous example shows that not only free HO variables must be avoided in expressions, but also free variables with *unsafe* types as *BoolFunctContainer*. The reason is that patterns with unsafe types may contain HO variables. Those patterns can appear in left-hand sides of rules, so a narrowing step can unify a free variable with one of these patterns, thereby introducing free HO variables—notice that the unification of $X$ and *bfc* $F_1$ introduces the free HO variable $F_1$ in the previous example. To formalize these intuitions we define the set of *unsafe* types as those for which problematic patterns can be formed:

DEFINITION 4.3 (Unsafe types). *The set of unsafe types wrt. a set of assumptions $\mathcal{A}$ ($UTypes_{\mathcal{A}}$) is defined as the least set of simple types verifying:*

1. *Functional types ($\tau \rightarrow \tau'$) are in $UTypes_{\mathcal{A}}$.*
2. *A simple type $\tau$ is in $UTypes_{\mathcal{A}}$ if there exists some pattern $t \in Pat$ with $\{\overline{X_n}\} = var(t)$ such that:*
   - *a) $t \equiv \mathcal{C}[X_i]$ with $\mathcal{C} \neq [\ ]$*
   - *b) $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$, for some $\overline{\tau_n}$*
   - *c) $\tau_i \in UTypes_{\mathcal{A}}$.*

$$(\Lambda^r)\ \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \qquad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \oplus \{\overline{Y_k : \tau'_k}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda^r t.e : \tau_t \rightarrow \tau}$$

where $\{\overline{X_n}\} = var(t)$, $\{\overline{Y_k}\} = fv(\lambda^r t.e)$ such that $\overline{\tau'_k}$ are ground and safe wrt. $\mathcal{A}$.

**Figure 4.** Typing rule for restricted $\lambda$-abstractions

For brevity we say a variable $X$ is *unsafe* wrt. $\mathcal{A}$ if $\mathcal{A}(X)$ is unsafe wrt. $\mathcal{A}$.

Clearly, if an expression does not contain free unsafe variables it does not contain free HO variables either, so by Lemma 4.1 neither (VAct) nor (VBind) could be used in a narrowing step. However, the absence of unsafe variables is not preserved after $\rightsquigarrow^{lmgu}$ steps even if the rules do not contain unsafe extra variables:

EXAMPLE 4.4. *Consider the symbols in Example 4.2 and a new function $g$ defined as $\{g \rightarrow X\}$ with type $g : \forall\alpha.\alpha$. The extra variable $X$ has the polymorphic type $\alpha$ in the rule for $g$, so it is safe. The expression $(f\ g)$ does not contain any unsafe variable, however we can make the reduction:*

$$f\ g \rightsquigarrow^{lmgu}_{\epsilon} f\ X_1 \rightsquigarrow^{lmgu}_{[X_1 \mapsto bfc\ F_1]} F_1\ true$$

*The new variable $X_1$ introduced has type $BoolFunctContainer$, which is unsafe.*

Example 4.4 shows that not only unsafe free variables must be avoided, but any expression of unsafe type which can be reduced to a free variable. In this case the problematic expression is $g$, which has type *BoolFunctContainer* and produces a free variable. Example 4.4 also shows that polymorphic extra variables are a source of problems, since they can take unsafe types depending on each particular use.

### 4.2 Restricted programs

Based on the problems detected in the previous section, we characterize a restricted class of programs and expressions to evaluate in which $\rightsquigarrow^{lwt}$ steps do not apply (VAct) and (VBind). First, we need that the expression to evaluate does not contain unsafe variables. Second, we forbid rules whose extra variables have unsafe types. Finally, we must also avoid polymorphic extra variables, since they can take different types, in particular unsafe ones. The restriction over programs is somehow tight: any program with functions using polymorphic extra variables are out of this family of programs, in particular the function *sublist* in Section 1 and other common functions using extra variables—see Section 4.3 for a detailed discussion.

In order to define formally this family of programs, we propose a restricted notion of well-typed programs. This notion is very similar to that in Definition 3.1, but using the restricted typing rule $(\Lambda^r)$ for $\lambda$-abstractions in Figure 4, which avoids extra variables with polymorphic or unsafe types.

DEFINITION 4.5 (Well-typed restricted program). *A program rule $f \rightarrow e$ is well-typed restricted wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ where $\mathcal{A}(f) \succ_{var} \tau$, $\{\overline{X_n}\} = fv(e)$ and $\overline{\tau_n}$ are some ground and* *safe simple types wrt. $\mathcal{A}$. A program rule $(f\ \overline{p_n} \rightarrow e)$ (with $n > 0$) is well-typed restricted wrt. $\mathcal{A}$ iff $\mathcal{A} \vdash \lambda^r p_1 \ldots \lambda^r p_n.e : \tau$ with $\mathcal{A}(f) \succ_{var} \tau$. A program $\mathcal{P}$ is well-typed restricted wrt. $\mathcal{A}$ if all its rules are well-typed restricted wrt. $\mathcal{A}$.*

If a program $\mathcal{P}$ is well-typed restricted wrt. $\mathcal{A}$ we write $wt^r_{\mathcal{A}}(\mathcal{P})$. Notice that for any $\mathcal{P}$ and $\mathcal{A}$ we have that $wt^r_{\mathcal{A}}(\mathcal{P})$ implies $wt_{\mathcal{A}}(\mathcal{P})$. For the rest of the section we will implicitly use this notion of well-typed restricted programs. Since the notion of well-typed substitution, and as a consequence the notion of $\rightsquigarrow^{lwt}$ step, is parameterized by the type system, then further mentions to $\rightsquigarrow^{lwt}$ in this section will refer to a relation slightly smaller than the one presented in Section 3.3: a variant of $\rightsquigarrow^{lwt}$ based on the type system from Definition 4.5. It is easy to see that this variant also preserves types in derivations. Therefore, although the following results are limited to this variant, they are still relevant.

The key property of well-typed restricted programs is that, starting from an expression without unsafe variables, the resulting expression of a $\rightsquigarrow^{lwt}$ reduction do not contain such variables either:

LEMMA 4.6 (Absence of unsafe variables). *Let $e$ be an expression not containing unsafe variables wrt. $\mathcal{A}$ and $\mathcal{P}$ be a program such that $wt^r_{\mathcal{A}}(\mathcal{P})$. If $e \rightsquigarrow^{lwt^*}_{\theta} e'$ then $e'$ does not contain unsafe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

Notice that the use of mgu's in the $\rightsquigarrow^{lwt}$ steps is not necessary in the previous lemma, as the absence of unsafe variables is guaranteed by the well-typed substitution implicit in the definition of the $\rightsquigarrow^{lwt}$. Based on Lemma 4.6, it is easy to prove that $\rightsquigarrow^{lmgu}$ is complete to the restriction of $\rightsquigarrow^{lwt}$ to mgu's:

THEOREM 4.7 (Completeness of $\rightsquigarrow^{lmgu}$ wrt. $\rightsquigarrow^{lwt}$). *Let $e$ be an expression not containing unsafe variables wrt. $\mathcal{A}$ and $\mathcal{P}$ be a program such that $wt^r_{\mathcal{A}}(\mathcal{P})$. If $e \rightsquigarrow^{lwt^*}_{\theta} e'$ using mgu's in each step then $e \rightsquigarrow^{lmgu^*}_{\theta} e'$.*

Notice that completeness is assured even for programs having non transparent left-hand sides, as well-typedness of substitutions is guaranteed by $\rightsquigarrow^{lwt}$.

### 4.3 Expressiveness of the restricted programs

The previous section states the completeness of $\rightsquigarrow^{lmgu}$ wrt. $\rightsquigarrow^{lwt}$ for the class of well-typed restricted programs, when only mgu's are used in (Narr) steps. However this class leaves outside a number of interesting functions containing extra variables. For example, the *sublist* function in Section 1 is discarded. The reason is that extra variables of the rule—*Us* and *Vs*—must have type $[\alpha]$, which is not ground. A similar situation happens with other well-known polymorphic functions using extra variables, as the *last* function to compute the last element of a list—$last\ Xs \rightarrow cond\ (Ys +\!+ [E] == Xs)\ E$ [15]—or the function to compute the inverse of a function at some point—$inv\ F\ X \rightarrow cond\ (F\ Y == X)\ Y$. A consequence is that the class of well-typed restricted programs excludes many polymorphic functions using extra variables, since they usually have extra variables with polymorphic types.

However, not all functions using extra variables are excluded from the family of well-typed restricted programs. An example is the *even* function from Section 1 that checks whether a natural number is even or not. The whole rule has type $nat \rightarrow nat$ and it contains the extra variable $Y$ of type *nat*, which is ground and safe, making the rule valid. Other functions handling natural numbers and using extra variables as $compound\ X \rightarrow cond\ (times\ M\ N == X)\ true$—where *times* computes the product of natural numbers—are also valid, since both $M$ and $N$ have type *nat*. Moreover, versions of the rejected polymorphic functions adapted to concrete ground types are also in the family of well-typed restricted programs. For example, functions as *sublistNat* or *lastBool* with types $[nat] \rightarrow [nat] \rightarrow bool$ and $[bool] \rightarrow bool$ and the same rules as their polymorphic versions are accepted. However, this is not a satisfactory solution: the generation of versions for the different types used implies duplication of code, which is clearly contrary to the degree of code reuse and generality offered by declarative languages—specially by means of polymorphic functions and the different input/output modes of function arguments.

The class of well-typed restricted programs is tighter than desired, and leaves out several interesting functions. Furthermore, for some of those functions—as *subslist* or *last*—we have not discovered any example where unsafe variables were introduced during reduction[4]. Therefore, we plan to further investigate the characterization of such a family in order to widen the number of programs accepted, while leaving out the problematic ones.

## 5. Type Preservation for Needed Narrowing

In this section we consider the type preservation problem for a simplified version of the Curry language, where features irrelevant to the scope of this paper are ignored, like constraints, encapsulated search, i/o, etc. Therefore we restrict ourselves to *simple Curry programs*, i.e., programs using only first-order patterns and transparent constructor symbols—which implies that all the patterns in left-hand sides are transparent. Besides, programs will be evaluated using the *needed narrowing* strategy [5] and performing residuation for variable applications—which is simulated by dropping the rules (VAct) and (VBind). We have decided to focus on needed narrowing because it is the most popular on-demand evaluation strategy, and it is at the core of the majority of modern FLP systems.

We use a transformational approach to employ $\rightsquigarrow^{lmgu}$ to simulate an adaptation of the needed narrowing strategy for let-narrowing. We rely on two program transformations well-known in the literature. In the first one, we start with an arbitrary simple Curry program and transform it into an *overlapping inductively sequential* (OIS) program [1]. For programs in this class, an *overlapping definitional tree* is available for every function, that encodes the demand structure implied by the left-hand sides of its rules. Then we proceed with the second transformation, which takes an OIS program and transforms it into *uniform format* [31]: programs in which the left-hand sides of the rules for every function $f$ have either the shape $f\ \overline{X}$ or $f\ \overline{X}\ (c\ \overline{Y})\ \overline{Z}$.

There are other well-known transformations from general programs to OIS programs—for example [10]—but we have chosen the transformation in Definition 5.1—which is similar to the transformation in [2], but now extended to generate type assumptions—because of its simplicity. The transformation processes each function independently: it takes the set of rules $\mathcal{P}_f$ for each function $f$ and returns a pair composed by the transformed rules and a set of assumptions for the auxiliary fresh functions introduced by the transformation.

DEFINITION 5.1 (Transformation to OIS). *Let $\mathcal{P}_f \equiv \{f\ \overline{t^1_n} \rightarrow e^1, \ldots, f\ \overline{t^m_n} \rightarrow e^m\}$ be a set of $m$ program rules for the function $f$ such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $f$ is an OIS function, $OIS(\mathcal{P}_f) = (\mathcal{P}_f, \emptyset)$. Otherwise $OIS(\mathcal{P}_f) = (\{f_1\ \overline{t^1_n} \rightarrow e^1, \ldots, f_m\ \overline{t^m_n} \rightarrow e^m, f\ \overline{X_n} \rightarrow f_1\ \overline{X_n} ? \ldots ? f_m\ \overline{X_n}\}, \{\overline{f_m : \mathcal{A}(f)}\})$, where $?$ is the non-determistic choice function defined with the rules $\{X?Y \rightarrow X, X?Y \rightarrow Y\}$.*

[4] The function *inv* can introduce HO variables when combined with a constant function as $zero\ X \rightarrow z$ with type $\forall\alpha.\alpha \rightarrow nat$: $(inv\ zero\ z)\ true \rightsquigarrow^{lwt^*}_{\theta} Y_1\ true$, where $Y_1$ is clearly unsafe.

The following result states that the transformation $OIS$ preserves types. Notice that any other transformation to OIS format that also preserves types could be used instead.

THEOREM 5.2 ($OIS(\mathcal{P}_f)$ well-typedness). *Let $\mathcal{P}_f$ be a set of program rules for the same function $f$ such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $OIS(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ then $wt_{\mathcal{A} \oplus \mathcal{A}'}(\mathcal{P}')$.*

After the transformation the assumption for $f$ remains the same and the new assumptions refer to fresh function symbols. Therefore, it is easy to see that the previous result is also valid for programs with several functions.

Now, to transform the program from OIS into uniform format we use the following transformation, which is a slightly variant of the transformation in [31]. Like in the previous transformation, we treat each function independently, returning the translated rules together with the extra assumptions for the auxiliary functions.

DEFINITION 5.3 (Transformation to uniform format). *Let $\mathcal{P}_f \equiv \{f\ \overline{t_n^1} \to e^1, \ldots, f\ \overline{t_n^m} \to e^m\}$ be an OIS program of $m$ program rules for a function $f$ such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $\mathcal{P}_f$ is already in uniform format, then $\mathcal{U}(\mathcal{P}_f) = (\mathcal{P}', \emptyset)$. Otherwise, we take the uniformly demanded position*[5] *$o$ and split $\mathcal{P}_f$ into $r$ sets $\overline{\mathcal{P}_r}$ containing the rules in $\mathcal{P}_f$ with the same constructor symbol in position $o$. Then $\mathcal{U}(\mathcal{P}_f) = (\bigcup_{i=1}^{r} \mathcal{P}_i' \cup \mathcal{P}'', \bigcup_{i=1}^{r} \mathcal{A}_i' \cup \mathcal{A}'')$ where:*

- $\mathcal{U}(\mathcal{P}_i^o) = (\mathcal{P}_i', \mathcal{A}_i')$
- *$c_i$ is the constructor symbol in position $o$ in the rules of $\mathcal{P}_i$, with $ar(c_i) = k_i$*
- *$\mathcal{P}_i^o$ is the result of replacing the function symbol $f$ in $\mathcal{P}_i$ by $f_{(c_i,o)}$ and flattening the patterns in position $o$ in the rules, i.e., $f\ \overline{t_j}\ (c_i\ \overline{t_{k_i}'})\ \overline{t_l''} \to e$ is replaced by $f_{(c_i,o)}\ \overline{t_j}\ \overline{t_{k_i}'}\ \overline{t_l''} \to e$*
- *$\mathcal{P}'' \equiv \{f\ \overline{X_j}\ (c_1\ \overline{Y_{k_1}})\ \overline{Z_l} \to f_{(c_1,o)}\ \overline{X_j}\ \overline{Y_{k_1}}\ \overline{Z_l}, \ldots,$ $f\ \overline{X_j}\ (c_r\ \overline{Y_{k_r}})\ \overline{Z_l} \to f_{(c_r,o)}\ \overline{X_j}\ \overline{Y_{k_r}}\ \overline{Z_l}\}$, with $\overline{X_j}\ \overline{Y_{k_i}}\ \overline{Z_l}$ pairwise distinct fresh variables such that $j + l + 1 = n$*
- *$\mathcal{A}'' \equiv \{f_{(c_1,o)} : \forall\overline{\alpha}.\overline{\tau_j} \to \overline{\tau_{k_1}'} \to \overline{\tau_l} \to \tau, \ldots, f_{(c_r,o)} : \forall\overline{\alpha}.\overline{\tau_j} \to \overline{\tau_{k_r}'} \to \overline{\tau_l} \to \tau\}$ where $\mathcal{A}(f) = \forall\overline{\alpha}.\overline{\tau_j} \to \tau' \to \overline{\tau_l} \to \tau$ and $\mathcal{A} \oplus \{\overline{Y_{k_i} : \tau_{k_i}'}\} \vdash c_i\ \overline{Y_{k_i}} : \tau'$. Notice that since constructor symbols $c_i$ are transparent, these $\overline{\tau_{k_i}'}$ do exist and are univocally fixed.*

This transformation also preserves types. For the same reasons as before, the following result is also valid for programs with several functions.

THEOREM 5.4 ($\mathcal{U}(\mathcal{P}_f)$ well-typedness). *Let $\mathcal{P}_f$ be a set of program rules for the same overlapping inductive sequential function $f$ such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $\mathcal{U}(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ then $wt_{\mathcal{A} \oplus \mathcal{A}'}(\mathcal{P}')$.*

We have just seen that we can transform an arbitrary program into uniform format while preserving types. The preservation of the semantics is also stated in [2, 31]. Although these results have been proved in the context of term rewriting, we strongly believe that they remain valid for the call-time choice semantics of the HO-CRWL framework. Similarly, we are strongly confident that the completeness of narrowing with mgu's over a uniform program wrt. needed narrowing over the original program [31] is also valid in the framework of let-narrowing. Combining those results with the type preservation results for $\rightsquigarrow^{lmgu}$ and the program transformations—Theorems 3.11, 5.2 and 5.4—we can conclude that a simulation of the evaluation of simple Curry programs using $\rightsquigarrow^{lmgu}$ based on the transformations above, is safe wrt. types.

## 6. Conclusions and Future Work

In this paper we have tackled the problem of type preservation for FLP programs with extra variables. As extra variables lead to the introduction of fresh free variables during the computations, we have decided to use the let-narrowing relation $\rightsquigarrow^{l}$—which is sound and complete wrt. HO-CRWL, a standard semantics for FLP—as the operational mechanism for this paper. This is also a natural choice because let-narrowing reflects the behaviour of current FLP systems like Toy or Curry, that provide support for extra and logical variables instead of reducing expressions by rewriting only.

The other main technical ingredient of the paper is a novel variation of Damas-Milner type system that has been enhanced with support for extra variables. Based on this type system we have defined the well-typed let-narrowing relation $\rightsquigarrow^{lwt}$, which is a restriction of let-narrowing that preserves types. To the best of our knowledge, this is the first paper proposing a polymorphic type system for FLP programs with logical and extra variables such that type preservation is formally proved. As we have seen in Example 3.2 from Section 3 the type systems from [21, 22] lose type preservation when extra variables are introduced. In [4], another remarkable previous work, the proposed type system only supports monomorphic functions and extra variables are not allowed. In [14] only programs with transparent patterns and without extra variables are considered, and functional arguments in data constructors are forbidden. Nevertheless, any of those programs is supported by our $\rightsquigarrow^{lwt}$ relation, which has to carry type information at run-time, but just like the extension of the Constructor-based Lazy Narrowing Calculus proposed in [14].

The relevance of Theorem 3.8, which states that $\rightsquigarrow^{lwt}$ preserves types, lies in the clarification it makes of the problem of type preservation on narrowing reductions with programs with extra variables. Relying on the abstract notion of well-typed substitution, which is parametrized by the type system and independent of any concrete operational mechanism, we have isolated a sufficient condition that ensures type preservation when imposed to the unifiers used in narrowing derivations. This contrasts with previous works like [14]—the closest to the present paper—in which a most general unifier was implicitly computed. Moreover, $\rightsquigarrow^{lwt}$ preserves types for arbitrary programs, something novel in the field of type systems in FLP—to the best of our knowledge. Hence, $\rightsquigarrow^{lwt}$ is an intended ideal narrowing relation that always preserves types, but that can only be directly realized by using type checks at run-time. Therefore, $\rightsquigarrow^{lwt}$ is most useful when used as a reference to define some imperfect but more practical materializations of it—subrelations of $\rightsquigarrow^{lwt}$—that only work for certain program classes but also preserve types while avoiding run-time type checks. An example of this is the relation $\rightsquigarrow^{lmgu}$, whose applicability is restricted to programs with transparent patterns, and that also lacks some completeness. This relation is based on two conditions imposed over $\rightsquigarrow^{l}$ steps: mgu's are used in every (Narr) step; and the rules (VAct) and (VBind) are avoided. While the former is not a severe restriction—as $\rightsquigarrow^{l}$ is complete wrt. HO-CRWL even if only mgu's are allowed as unifiers [24]—the latter is more problematic, because then $\rightsquigarrow^{lmgu}$ is not able to generate bindings for variable applications. To mitigate this weakness we have investigated how to prevent the use of (VAct) and (VBind) in $\rightsquigarrow^{lwt}$ derivations. After some preliminary attempts that witness the difficulty of the task, and also give valuable insights about the problem, we have finally characterized a class of programs in which these bindings for variable applications are not needed, and studied their expressiveness. Then we have applied the results obtained so far for proving the type preservation for a simplified version of the Curry language. HO-patterns are not supported in Curry, which treats functions as black boxes [4]. Therefore Curry programs do not intend to gen-

[5] A position in which all the rules in $\mathcal{P}_f$ have a constructor symbol. Notice that this position will always exist because $\mathcal{P}_f$ is an OIS program [1].

erate solutions that include bindings for variable applications, and so the rules (VAct) and (VBind) will not be used to evaluate these programs. Besides, in Curry all the constructors are transparent, and the needed narrowing on-demand strategy is employed in most implementations of Curry. We have used two well-known program transformations to simulate the evaluation of Curry programs with an adaptation of needed narrowing for let-narrowing. Then we have proved that both transformations preserve types which, combined with the type preservation of $\leadsto^{lmgu}$, implies that our proposed simulation of needed narrowing also preserves types.

Regarding future work, we would like to look for new program classes more general than the one presented in Section 4 because, as we pointed out at the end of that section, the proposed class is quite restrictive and it forbids several functions that we think are not dangerous for the types.

Another interesting line of future work would deal with the problems generated by opaque pattens, as we did in [22] for the restricted case where we drop logical and extra variables. We think that an approach in the line of existential types [20] that, contrary to [22], forbids pattern matching over existential arguments, is promising. This has to do with the parametricy property of types systems [30], which is broken in [22] as we allowed matching on existential arguments, and which is completely abandoned from the very beginning in [21]. In fact it was already detected in [14] that the loss of parametricity leads to the loss of type preservation in narrowing derivations—in that paper instead of parametricity the more restrictive property of type generality is considered. All that suggests that our first task regarding this subject should be modifying our type system from [22] to recover parametricity by following an approach to opacity closer to standard existential types.

## Acknowledgments

This work has been partially supported by the Spanish projects FAST-STAMP (TIN2008-06622-C03-01), PROMETIDOS-CM (S2009TIC-1465) and GPD-UCM (UCM-BSCH-GR35/10-A-910502).

## References

[1] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. 6th Int. Conf. on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.

[2] S. Antoy. Constructor based conditional narrowing. In *Proc. 3rd Int. Conf. on Principles and Practice of Declarative Programming (PPDP'01)*, pages 199–206. ACM, 2001.

[3] S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, 2010.

[4] S. Antoy and A. Tolmach. Typed higher-order narrowing without higher-order strategies. In *Proc. 4th Int. Symp. on Functional and Logic Programming (FLOPS'99)*, pages 335–352. Springer LNCS 1722, 1999.

[5] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47:776–822, July 2000.

[6] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[7] B. Brassel. Two to three ways to write an unsafe type cast without importing unsafe - Curry mailing list. `http://www.informatik.uni-kiel.de/~curry/listarchive/0705.html`, May 2008.

[8] B. Brassel, S. Fischer, M. Hanus and F. Reck Transforming Functional Logic Programs into Monadic Functional Programs In *Proc. 19th Int. Work. on Functional and (Constraint) Logic Programming (WFLP'10)*, Springer LNCS 6559, pages 30–47, 2011.

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

[10] R. del Vado Vírseda. Estrategias de estrechamiento perezoso. Master's thesis, Universidad Compluetense de Madrid, 2002.

[11] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard. Reference Manual*. Springer, 1996.

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

[13] 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.

[14] 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.

[15] M. Hanus. Multi-paradigm declarative languages. In *Proc. 23rd Int. Conf. on Logic Programming (ICLP'07)*, pages 45–75. Springer LNCS 4670, 2007.

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

[17] M. Hanus and F. Steiner. Type-based nondeterminism checking in functional logic programs. In *Proc. 2nd. Inf. Conf. Principles and Practice of Declarative Programming. (PDP 2000)*, pages 202–213, ACM, 2000.

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

[19] J.-M. Hullot. Canonical forms and unification. In *Proc. 5th Conf. on Automated Deduction (CADE-5)*, pages 318–334. Springer LNCS 87, 1980.

[20] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16:1411–1430, 1994.

[21] F. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. Liberal typing for functional logic programs. In *Proc. 8th Asian Symp. on Programming Languages and Systems (APLAS'10)*, pages 80–96. Springer LNCS 6461, 2010.

[22] F. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. New results on type systems for functional logic programming. In *Proc. 18th Int. Workshop on Functional and (Constraint) Logic Programming (WFLP'09), Revised Selected Papers*, pages 128–144. Springer LNCS 5979, 2010.

[23] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. 10th Int. Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[24] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th Int. Symp. on Functional and Logic Programming (FLOPS'08)*, pages 147–162. Springer LNCS 4989, 2008.

[25] 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.

[26] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

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

[28] E. Martin-Martin. Type classes in functional logic programming. In *Proc. 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'11)*, pages 121–130. ACM, 2011.

[29] M. Rodríguez-Artalejo. Functional and constraint logic programming. In *Constraints in Computational Logics*, pages 202–270. Springer LNCS 2002, 2001.

[30] P. Wadler. Theorems for free! In *Proc. 4th Int. Conf. on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 347–359. ACM, 1989.

[31] F. Zartmann. Denotational abstract interpretation of functional logic programs. In *Proc. 4th Int. Symp. on Static Analysis (SAS'97)*, pages 141–159. Springer LNCS 1302, 1997.

# A. Proofs

The following theorem contains some interesting properties of the typing relation $\vdash$ in Figure 3 that will be used intensively in this appendix. [27] contains detailed proofs for these properties for a very similar type relation whose $(\Lambda)$ rule does not handle $\lambda$-abstractions with extra variables. However, the extension of those proofs to support the new flavour of $\lambda$-abstractions is straightforward and has been omitted.

THEOREM A.1 (Properties of the typing relation).

*a) If $\mathcal{A} \vdash e : \tau$ then $\mathcal{A}\pi \vdash e : \tau\pi$, for any $\pi \in TSubst$.*
*b) Let $s$ be a symbol not occurring in $e$. Then $\mathcal{A} \vdash e : \tau \iff \mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$, for any $\sigma$.*
*c) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash e : \tau$ and $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$ then $\mathcal{A} \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$.*

### A.1 Proof of Theorem 3.8: Type preservation of $\rightsquigarrow^{lwt}$

In order to prove Type Preservation, we need the following auxiliary result regarding type preservation with contexts and well-typed substitutions:

LEMMA A.2. *Consider $\mathcal{A} \vdash \mathcal{C}[e] : \tau$ containing the subderivation $\mathcal{A} \oplus [\overline{Z_m/\tau_m}] \vdash e : \tau_e$ (being $[\overline{Z_m/\tau_m}]$ the set of assumptions generated for bound variables) and $\mathcal{A} \oplus [\overline{Z_m/\tau_m}] \vdash e' : \tau_e$. Define $\mathcal{A}_0 \equiv \mathcal{A}$ and $\mathcal{A}_i \equiv \mathcal{A}_{i-1} \oplus \{Z_i : \tau_i\}$ for $i \in [1..m]$. In that conditions, if we have a data derivation $\theta$ such that $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C})})$ for every $i \in [0..m]$ and $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$ then $\mathcal{A} \vdash \mathcal{C}\theta[e'] : \tau$.*

**Proof** By induction on the structure of $\mathcal{C}$.

BASE CASE:

$\boxed{\mathcal{C} \equiv [\,]}$ In this case $\mathcal{A}_m \equiv \mathcal{A} \oplus [\overline{Z_m/\tau_m}]$, so $\mathcal{A}_m \vdash \mathcal{C}[e] : \tau$ with $\mathcal{C}[e] \equiv e$ and $\tau \equiv \tau_e$. By hypothesis we have $\mathcal{A} \oplus [\overline{Z_m/\tau_m}] \vdash e : \tau$, so $\mathcal{A} \oplus [\overline{Z_m/\tau_m}] \vdash \mathcal{C}[e']$.

INDUCTIVE STEP:

$\boxed{\mathcal{C} \equiv \mathcal{C}'\ e_2}$ In this case we have

$$\text{(APP)}\,\frac{\mathcal{A}_n \vdash \mathcal{C}'[e] : \tau' \to \tau \qquad \mathcal{A}_n \vdash e_2 : \tau'}{\mathcal{A}_n \vdash \mathcal{C}'[e]\ e_2 : \tau}$$

for a $\mathcal{A}_n$ containing assumptions for the bound variables reached up to this point. By the hypothesis we have that $wt_{\mathcal{A}_n}(\theta|_{fv(\mathcal{C})})$, so for any free variable $X \in e_2$ the substitution $\theta$ verifies $\mathcal{A}_n \vdash X\theta : \mathcal{A}_n(X)$. Then by Theorem A.1-c) we have $\mathcal{A}_n \vdash e_2\theta : \tau'$. From the hypothesis we know that the derivation $\mathcal{A}_n \vdash \mathcal{C}'[e] : \tau' \to \tau$ contains a subderivation $\mathcal{A} \oplus [\overline{Z_m/\tau_m}] \vdash e : \tau_e$ and $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C})})$ for any $i \in [n..m]$, so $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C}')})$ for any $i \in [n..m]$ as $fv(\mathcal{C}') \subseteq fv(\mathcal{C})$. From the hypothesis we also have $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$, so $dom(\theta) \cap bv(\mathcal{C}') = \emptyset$ since $bv(\mathcal{C}'\ e_2) = bv(\mathcal{C}')$. Then by the Induction Hypothesis $\mathcal{A}_n \vdash \mathcal{C}'\theta[e'] : \tau' \to \tau$ and since $\mathcal{C}'\theta[e']\ e_2\theta \equiv \mathcal{C}\theta[e']$ we have:

$$\text{(APP)}\,\frac{\mathcal{A}_n \vdash \mathcal{C}'[e'] : \tau' \to \tau \qquad \mathcal{A}_n \vdash e_2\theta : \tau'}{\mathcal{A}_n \vdash \mathcal{C}\theta[e'] : \tau}$$

$\boxed{\mathcal{C} \equiv \mathcal{C}'\ e_2}$ Similar to the previous case.

$\boxed{\mathcal{C} \equiv let\ Z_n = \mathcal{C}'\ in\ e_2}$ We have a derivation

$$\text{(LET)}\,\frac{\mathcal{A}_n \vdash \mathcal{C}'[e] : \tau_n \qquad \mathcal{A}_{n+1} \vdash e_2 : \tau}{\mathcal{A}_n \vdash let\ Z_n = \mathcal{C}'[e]\ in\ e_2 : \tau}$$

where $\mathcal{A}_n$ contains assumptions for the bound variables reached up to this point and $\mathcal{A}_{n+1} \equiv \mathcal{A}_n \oplus \{Z_n : \tau_n\}$ by definition. By the hypothesis we have that $wt_{\mathcal{A}_{n+1}}(\theta|_{fv(\mathcal{C})})$, so for any free variable $X \in e_2$ the substitution $\theta$ verifies $\mathcal{A}_{n+1} \vdash X\theta : \mathcal{A}_{n+1}(X)$. Then by Theorem A.1-c) we have $\mathcal{A}_{n+1} \vdash e_2\theta : \tau$. From the hypothesis we know that the derivation $\mathcal{A}_n \vdash \mathcal{C}'[e] : \tau_n$ contains a subderivation $\mathcal{A} \oplus [\overline{Z_m/\tau_m}] \vdash e : \tau_e$ and $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C})})$ for any $i \in [n..m]$, so $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C}')})$ for any $i \in [n..m]$ as $fv(\mathcal{C}') \subseteq fv(\mathcal{C})$. Also from the hypothesis we have have $dom(\theta) \cap bv(let\ Z_n = \mathcal{C}'\ in\ e_2) = \emptyset$, so $dom(\theta) \cap bv(\mathcal{C}') = \emptyset$ since $bv(let\ Z_n = \mathcal{C}'\ in\ e_2) = bv(\mathcal{C}')$. Then by the Induction Hypothesis $\mathcal{A}_n \vdash \mathcal{C}'\theta[e'] : \tau_n$, and considering that $let\ Z_n = \mathcal{C}'\theta[e']\ in\ e_2\theta \equiv \mathcal{C}\theta[e']$ we have:

$$\text{(LET)}\,\frac{\mathcal{A}_n \vdash \mathcal{C}'\theta[e'] : \tau_n \qquad \mathcal{A}_{n+1} \vdash e_2\theta : \tau}{\mathcal{A}_n \vdash \mathcal{C}\theta[e'] : \tau}$$

$\boxed{\mathcal{C} \equiv let\ Z_n = e_1\ in\ \mathcal{C}'}$ Similar to the previous case, with two main differences. The first one is that $dom(\theta) \cap bv(\mathcal{C}') = \emptyset$ because $bv(\mathcal{C}') \subseteq bv(let\ Z_n = e_1\ in\ \mathcal{C}')$. The second difference is that $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C}')})$ for any $i \in [n+1..m]$ because $wt_{\mathcal{A}_i}(\theta|_{(fv(\mathcal{C}') \smallsetminus \{Z_n\})})$ for any $i \in [n+1..m]$ as $fv(\mathcal{C}') \smallsetminus \{Z_n\} \subseteq fv(\mathcal{C})$, and using the fact that $Z_n \notin dom(\theta)$—since $Z_n \in bv(\mathcal{C})$—then $\theta|_{(fv(\mathcal{C}') \smallsetminus \{Z_n\})} \equiv \theta|_{(fv(\mathcal{C}')}$

Using the previous lemma, we can now prove Type Preservation:

**Theorem 3.8 (Type preservation of $\rightsquigarrow^{lwt}$)**
*If $wt_{\mathcal{A}}(\mathcal{P})$, $e \rightsquigarrow_{\theta}^{lwt^*} e'$ and $\mathcal{A} \vdash e : \tau$ then $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$ and $wt_{\mathcal{A} \oplus \mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

**Proof** We first prove the result for one step $e \rightsquigarrow_{\theta}^{lwt} e'$ by case distinction over the used rule. Notice that $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$ is true by the hypothesis $e \rightsquigarrow_{\theta}^{lwt^*} e'$, so we only have to prove $\mathcal{A}\oplus\mathcal{A}' \vdash e' : \tau$. The proofs for the cases (LetIn), (Bind), (Elim), (Flat) and (LetAp) are the same as those in [27]. For the remaining cases:

- $\boxed{\text{(Narr)}}$

  For the sake of simplicity we will prove the case for a function applied to 2 patterns, but the proof for any number of arguments follows the same ideas. We have a narrowing step $f\ t_1\ t_2 \rightsquigarrow_{\theta}^{lwt} r\theta$ for a fresh variant $(f\ p_1\ p_2 \to r) \in \mathcal{P}$ and a well-typed substitution $\theta$ such that $(f\ p_1\ p_2)\theta \equiv (f\ t_1\ t_2)\theta$. From the hypothesis we have:

$$\text{(APP)}\dfrac{\text{(APP)}\dfrac{\mathcal{A}\vdash f:\tau_1\to\tau_2\to\tau \qquad \mathcal{A}\vdash t_1:\tau_1}{\mathcal{A}\vdash f\ t_1:\tau_2\to\tau} \qquad \mathcal{A}\vdash t_2:\tau_2}{\mathcal{A}\vdash f\ t_1\ t_2:\tau}$$

  Since the rule is well-typed, we also have a type derivation:

$$(\Lambda)\dfrac{\mathcal{A}\oplus\mathcal{A}_1\vdash p_1:\tau_1' \qquad (\Lambda)\dfrac{\begin{array}{c}\mathcal{A}\oplus\mathcal{A}_1\oplus\mathcal{A}_2\vdash p_2:\tau_2'\\ (A)\ \mathcal{A}\oplus\mathcal{A}_1\oplus\mathcal{A}_2\vdash r:\tau'\end{array}}{\mathcal{A}\oplus\mathcal{A}_1\vdash\lambda p_2.r:\tau_2'\to\tau'}}{\mathcal{A}\vdash\lambda p_1.\lambda p_2.r:\tau_1'\to\tau_2'\to\tau'}$$

  where $\mathcal{A}_1$ and $\mathcal{A}_2$ are assumptions over $var(p_1)\cup fv(\lambda p_1.\lambda p_2.r)$ and $var(p_2)\cup fv(\lambda p_2.r)$ resp. and $\tau_1'\to\tau_2'\to\tau'$ is a variant of $\mathcal{A}(f)$. Since $\tau_1\to\tau_2\to\tau$ is a generic instance of $\mathcal{A}(f)$ then $(\tau_1'\to\tau_2'\to\tau')\pi \equiv \tau_1\to\tau_2\to\tau$ for some type substitution $\pi$ whose domain are fresh type variables from the variant.
  By Theorem A.1-a) we can apply the type substitution $\pi$ to $(A)$:

$$(A')\ \mathcal{A}\oplus\mathcal{A}_1\pi\oplus\mathcal{A}_2\pi\vdash r:\tau$$

  noticing that $\tau'\pi \equiv \tau$ and $\mathcal{A}\pi \equiv \pi$ since the domain of $\pi$ are fresh type variables. The set of assumptions associated to this step is $\mathcal{A}' \equiv \mathcal{A}_1\pi\oplus\mathcal{A}_2\pi$, so by the premise $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$ and we can use Theorem A.1-c) to apply $\theta$ in $(A')$:

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

- $\boxed{\text{(VAct)}}$

  For the sake of conciseness, we consider the simplified step $X\ t_2 \rightsquigarrow_{\theta}^{lwt} r\theta$ for a fresh variant $(f\ p_1\ p_2 \to r) \in \mathcal{P}$ such that $(X\ t_2)\theta \equiv f\ p_1\theta\ p_2\theta$. From $wt_{\mathcal{A}}(e)$ we have:

$$\text{(APP)}\dfrac{\mathcal{A}\vdash X:\tau_2\to\tau \qquad \mathcal{A}\vdash t_2:\tau_2}{\mathcal{A}\vdash X\ t_2:\tau}$$

  Since $wt_{\mathcal{A}}(\mathcal{P})$ then the rule is well-typed, and we also have a type derivation:

$$(\Lambda)\dfrac{\mathcal{A}\oplus\mathcal{A}_1\vdash p_1:\tau_1' \qquad (\Lambda)\dfrac{\begin{array}{c}\mathcal{A}\oplus\mathcal{A}_1\oplus\mathcal{A}_2\vdash p_2:\tau_2'\\ (A)\ \mathcal{A}\oplus\mathcal{A}_1\oplus\mathcal{A}_2\vdash r:\tau'\end{array}}{\mathcal{A}\oplus\mathcal{A}_1\vdash\lambda p_2.r:\tau_2'\to\tau'}}{\mathcal{A}\vdash\lambda p_1.\lambda p_2.r:\tau_1'\to\tau_2'\to\tau'}$$

  where $\mathcal{A}_1$ and $\mathcal{A}_2$ are set of assumptions for the variables in $var(p_1)\cup fv(\lambda p_1.\lambda p_2.r)$ and $var(p_2)\cup fv(\lambda p_2.r)$ resp. Since the associated set of assumptions is defined by premise, we know that $\mathcal{A}' \equiv \mathcal{A}_1\pi\oplus\mathcal{A}_2\pi$ for some $\pi$ such that $(\tau_2'\to\tau')\pi \equiv \tau_2\to\tau$ and $fv(\mathcal{A})\cap dom(\pi) = \emptyset$. By Theorem A.1-a) we can apply the type substitution $\pi$ to $(A)$:

$$(A')\ \mathcal{A}\oplus\mathcal{A}_1\pi\oplus\mathcal{A}_2\pi\vdash r:\tau$$

  noticing that $\tau'\pi \equiv \tau$ and $\mathcal{A}\pi \equiv \pi$. By premise $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, so we can use Theorem A.1-c) to apply $\theta$ in $(A')$:

$$\mathcal{A}\oplus\mathcal{A}'\vdash r\theta:\tau$$

- $\boxed{\text{(VBind)}}$

  The step is $let\ X = e_1\ in\ e_2 \rightsquigarrow_{\theta}^{lwt} e_2\theta[X\mapsto e_1\theta]$, where $e_1 \notin Pat$, $e_1\theta \in Pat$ and $X \notin dom(\theta)\cup vran(\theta)$. From $wt_{\mathcal{A}}(e)$ we have:

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

  The set of assumptions $\mathcal{A}'$ associate to the step contains assumptions over the new variables introduced by $\theta$, so they cannot appear in $e_1$ or $e_2$. Then, by Theorem A.1-b) we can add them to $(A)$ and $(B)$:

$$\begin{array}{l}(A')\ \mathcal{A}\oplus\mathcal{A}'\vdash e_1:\tau_x\\ (B')\ \mathcal{A}\oplus\mathcal{A}'\oplus\{X:\tau_x\}\vdash e_2:\tau\end{array}$$

Since $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$ then by Theorem A.1-c) and $(A')$ we have

$$(A'')\ \mathcal{A}\oplus\mathcal{A}' \vdash e_1\theta : \tau_x$$

We can assume that $X \notin fv(e_1)$ since our let-expressions are not recursive. By the conditions of the step we know that $X \notin dom(\theta) \cup vran(\theta)$, so $X \notin e_1\theta$ and by Theorem A.1-b) we can add the assumption for $X$ to the derivation $(A'')$:

$$(A''')\ \mathcal{A}\oplus\mathcal{A}'\oplus\{X : \tau_x\} \vdash e_1\theta : \tau_x$$

Since $X \notin dom(\theta) \cup vran(\theta)$ then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$ implies $wt_{\mathcal{A}\oplus\mathcal{A}'\oplus\{X:\tau_x\}}(\theta)$, and by Theorem A.1-c) and $(B')$ we have:

$$(B'')\ \mathcal{A}\oplus\mathcal{A}'\oplus\{X : \tau_x\} \vdash e_2\theta : \tau$$

Finally, by A.1-c) and $(A''')$ we can apply the substitution $[X \mapsto e_1\theta]$ to $(B'')$:

$$(B''')\ \mathcal{A}\oplus\mathcal{A}'\oplus\{X : \tau_x\} \vdash e_2\theta[X \mapsto e_1\theta] : \tau$$

Since $e_2\theta[X \mapsto e_1\theta]$ does not contain $X$, by Theorem A.1-b) we can remove the assumption over it, obtaining:

$$\mathcal{A}\oplus\mathcal{A}' \vdash e_2\theta[X \mapsto e_1\theta] : \tau$$

- $\boxed{\text{(Contx)}}$

  We have a narrowing step $\mathcal{C}[e] \rightsquigarrow^{lwt}_{\theta} \mathcal{C}\theta[e']$ for $\mathcal{C} \neq [\,]$, $e \rightsquigarrow^{l}_{\theta} e'$ using any of the previous rules. By hypothesis we have $\mathcal{A} \vdash \mathcal{C}[e] : \tau$, so in this derivation there is a subderivation $\mathcal{A}\oplus\mathcal{A}_b \vdash e : \tau_e$ for some $\mathcal{A}_b \equiv \{\overline{Z_m : \tau_m}\}$ containing assumptions for the bound variables in $\mathcal{C}$.

  - If the step $e \rightsquigarrow^{lwt}_{\theta} e'$ uses a rule different from (Narr), (Vact) or (VBind), then $\theta \equiv \epsilon$ and by the proof of those cases $\mathcal{A}\oplus\mathcal{A}_b \vdash e' : \tau_e$ (since $\mathcal{A}' \equiv \emptyset$). Then by Lemma 6 in [27] we can replace an expression inside a context by any other of the same type, so $\mathcal{A} \vdash \mathcal{C}\epsilon[e'] : \tau$.
  - If the step $e \rightsquigarrow^{l}_{\theta} e'$ uses (Narr) or (VAct) then we have that *i)* $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$ and *ii)* the step uses a fresh variant $(f\ \overline{p_n} \rightarrow r) \in \mathcal{P}$ such that $vran(\theta|_{\setminus var(\overline{p_n})}) \cap bv(\mathcal{C}) = \emptyset$. We have $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$ by hypothesis and $\overline{Z_m}$ are bound variables which can be assumed not to appear in $\mathcal{A}$, so $wt_{\mathcal{A}\oplus\mathcal{A}_b\oplus\mathcal{A}'}(\theta)$. Therefore we have $e \rightsquigarrow^{lwt}_{\theta} e'$, and by the proof of one step we have $\mathcal{A}\oplus\mathcal{A}_b\oplus\mathcal{A}' \vdash e' : \tau$. The set $\mathcal{A}'$ contains assumptions over new data variables introduced in the step, and $\mathcal{A}_b$ contains assumptions over bound variables so $dom(\mathcal{A}_b) \cap dom(\mathcal{A}') = \emptyset$ and $\mathcal{A}\oplus\mathcal{A}_b\oplus\mathcal{A}' \vdash e' : \tau$ implies $\mathcal{A}\oplus\mathcal{A}'\oplus\mathcal{A}_b \vdash e' : \tau$. For the same reasons, $wt_{\mathcal{A}\oplus\mathcal{A}'\oplus\mathcal{A}_b}(\theta)$. As the variables in $\mathcal{A}'$ can appear neither in $e$ nor in $\mathcal{C}[e]$—and $dom(\mathcal{A}_b) \cap dom(\mathcal{A}') = \emptyset$—then by Theorem A.1-b) we have $\mathcal{A}\oplus\mathcal{A}'\oplus\mathcal{A}_b \vdash e : \tau_e$ and $\mathcal{A}\oplus\mathcal{A}' \vdash \mathcal{C}[e] : \tau$. Define $\mathcal{A}_0 \equiv \mathcal{A}\oplus\mathcal{A}'$ and $\mathcal{A}_i \equiv \mathcal{A}_{i-1}\oplus\{Z_i : \tau_i\}$ for any $i \in [1..m]$. From the fact that $\overline{p_n}$ are fresh variables and *ii)* we can conclude that $var(X\theta) \cap bv(\mathcal{C}) = \emptyset$ for every $X \in fv(\mathcal{C})$. We can assume that $bv(\mathcal{C}) \cap fv(\mathcal{C}) = \emptyset$, so by Theorem A.1-b) and $wt_{\mathcal{A}\oplus\mathcal{A}'\oplus\mathcal{A}_b}(\theta)$ it is clear that $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C})})$ for any $i \in [0..m]$. Finally, by Lemma A.2 we have that $\mathcal{A}\oplus\mathcal{A}' \vdash \mathcal{C}\theta[e'] : \tau$.
  - If the step $e \rightsquigarrow^{lwt}_{\theta} e'$ uses (VBind) then *i)* $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$ and *ii)* $vran(\theta) \cap bv(\mathcal{C}) = \emptyset$. The proof follows a similar reasoning to the previous case: from *ii)* and assuming $bv(\mathcal{C}) \cap fv(\mathcal{C}) = \emptyset$ we have $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C})})$ for any $i \in [0..m]$. Therefore by Lemma A.2 we have $\mathcal{A}\oplus\mathcal{A}' \vdash \mathcal{C}\theta[e'] : \tau$.

The proof for any number of steps proceeds by induction of the number of steps:
BASE CASE: $e \rightsquigarrow^{lwt^0}_{\epsilon} e'$
In this case $e \equiv e'$ and $\mathcal{A}' \equiv \emptyset$, so trivially $\mathcal{A} \vdash e' : \tau$ and $wt_{\mathcal{A}}(\epsilon)$.
INDUCTIVE STEP: $e \rightsquigarrow^{lwt^{n+1}}_{\theta_1\theta'} e' \equiv e \rightsquigarrow^{lwt}_{\theta_1} e_1 \rightsquigarrow^{lwt^n}_{\theta'} e'$
As $e \rightsquigarrow^{lwt^{n+1}}_{\theta_1\theta'} e'$, it is possible to check that there is a derivation $e \rightsquigarrow^{lwt^{n+1}}_{\theta_1\theta'} e'$ which uses type derivations $\mathcal{D}_i$ to $\tau$ in every inner step, so each set of assumptions $\mathcal{A}'_i$ associated to each step is related also to this derivation $\mathcal{D}_i$. By the proof of one step we have that $\mathcal{A}\oplus\mathcal{A}'_1 \vdash e_1 : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'_1}(\theta_1)$, where $\mathcal{A}'_1$ is the set of assumptions associated to the first step. Since the variables in $\mathcal{A}'_1$ cannot appear in $\mathcal{P}$, the program remains well-typed adding these new assumptions: $wt_{\mathcal{A}\oplus\mathcal{A}'_1}(\mathcal{P})$. Then by the Induction Hypothesis we have that $\mathcal{A}\oplus\mathcal{A}'_1\oplus\mathcal{A}'_n \vdash e' : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'_1\oplus\mathcal{A}'_n}(\theta')$, where $\mathcal{A}'_n$ is the set of assumptions associated to the reduction $e_1 \rightsquigarrow^{lwt^n}_{\theta'} e'$. The set $\mathcal{A}' \equiv \mathcal{A}'_1\oplus\mathcal{A}'_n$ contains assumptions over fresh variables. To prove $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta_1\theta')$ consider an arbitrary variable $X \in dom(\theta_1\theta')$:

- If $X \notin dom(\theta_1)$ then $X\theta_1\theta' \equiv X\theta'$ and $X \in dom(\theta')$. Trivially $\mathcal{A}\oplus\mathcal{A}' \vdash X\theta_1\theta' : (\mathcal{A}\oplus\mathcal{A}')(X)$ from $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta')$.
- If $X \in dom(\theta_1)$ then by $wt_{\mathcal{A}\oplus\mathcal{A}'_1}(\theta_1)$ we have $\mathcal{A}\oplus\mathcal{A}'_1 \vdash X\theta_1 : (\mathcal{A}\oplus\mathcal{A}'_1)(X)$. Since the variables in $\mathcal{A}'_n$ are fresh they do not occur in $X\theta_1$, so by Theorem A.1-b) $\mathcal{A}\oplus\mathcal{A}' \vdash X\theta_1 : (\mathcal{A}\oplus\mathcal{A}'_1)(X)$. Similarly, $X$ cannot appear in $\mathcal{A}'_n$, so $(\mathcal{A}\oplus\mathcal{A}'_1)(X) \equiv (\mathcal{A}\oplus\mathcal{A}')(X)$. Finally, since $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta')$ by Theorem A.1-c) we obtain $\mathcal{A}\oplus\mathcal{A}' \vdash X\theta_1\theta' : (\mathcal{A}\oplus\mathcal{A}')(X)$.

### A.2 Proof of Lemma 3.10: Mgu well-typedness

The proof uses a transformation approach ($\Longrightarrow$) similar to that presented in [6] to obtain mgu's—which follows the same ideas as the one in [26]. The difference is that our transformation does not orient equations prior to apply (Eliminate): it eliminates variables regardless of the side, giving priority to left-hand sides. This is important, since to prove well-typedness of mgu's we need that left-hand sides of equations remain transparent. However, it is easy to see that this transformation behaves the same as the original in [6].

$$\begin{array}{llll}
\text{(Delete)} & \{p =^? p\} \uplus S & \Longrightarrow & S \\
\text{(Decompose)} & \{h\,\overline{p_n} =^? h\,\overline{t_n}\} \uplus S & \Longrightarrow & \{p_1 =^? t_1, \ldots, p_n =^? t_n\} \cup S \\
\text{(EliminateL)} & \{X =^? t\} \uplus S & \Longrightarrow & \{X =^? t\} \cup S[X \mapsto t], \text{ if } X \in \mathit{fv}(S) \setminus var(t) \\
\text{(EliminateR)} & \{p =^? X\} \uplus S & \Longrightarrow & \{p =^? X\} \cup S[X \mapsto p], \text{ if } X \in \mathit{fv}(S) \setminus var(p) \text{ and } p \notin \mathcal{V}
\end{array}$$

The unification procedure $\mathcal{U}(p,t)$ starts with a set of one equation $\{p =^? t\}$ and performs $\Longrightarrow$ steps until it reaches normal form. If the normal form is in *solved form*—$\{\overline{X_n =^? t_n}\} \cup \{\overline{p_m =^? Y_m}\}$ where $p_i \notin \mathcal{V}$, $\{\overline{X_n}, \overline{Y_m}\}$ are pairwise distinct variables and $\{\overline{X_n}, \overline{Y_m}\} \cap (var(t_n) \cup var(p_m))$—the set represents the mgu $[\overline{X_n \mapsto t_n}, \overline{Y_m \mapsto p_m}]$, otherwise it fails.

In order to prove the well-typedness of mgu's obtained by $\mathcal{U}$ we need some extra results about the mentioned transition system. We use $\mathcal{U}$ to compute unifiers of left-hand sides of fresh variants of rules $f\,\overline{p_n}$ and expressions $f\,\overline{t_n}$. This particularity limits the sets of equations that we find along the computation of the mgu to *transparent sets*. To define transparent sets of equations we use the usual notion of *postions in expressions* $o \in \mathcal{O}$ [6], which are strings of positive integers using $\epsilon$ as the empty string. Then the *subexpression of e at position o*, denoted as $e|_o$, is defined as $e|_\epsilon = e$, $(h\ e_1\ \ldots\ e_n)|_{io} = e_i|_o$.

DEFINITION A.3 (Transparent set of equations). *We say a set of equations* $S \equiv \{\overline{p_n =^? t_n}\}$ *is* transparent *if every* $p_i$ *is transparent and if there exists an equation* $(p =^? t) \in S$ *and position* $o \in \mathcal{O}$ *such that* $p|_o \equiv X$ *and* $t|_o \equiv t'$ *with* $t'$ *a non-transparent pattern, then* $X$ *appears only once in the set of equations—exactly in that position of that equation.*

LEMMA A.4 ($\Longrightarrow$ steps preserve set transparency). *If* $S$ *is transparent and* $S \Longrightarrow^* S'$*, then* $S'$ *is also transparet.*

**Proof** The proof for one step proceeds by case distinction on the rule used:

- (Delete) Trivially.
- (Decompose) The step is $S \equiv \{h\,\overline{p_n} =^? h\,\overline{t_n}\} \uplus S'' \Longrightarrow \{p_1 =^? t_1, \ldots, p_n =^? t_n\} \cup S'' \equiv S'$. Since $h\,\overline{p_n}$ is a transparent pattern wrt. $\mathcal{A}$, the new patterns $\overline{p_n}$ introduced as left-hand sides are transparent as well. By premise, if there is a equation $(p =^? t) \in S''$ and position $o \in \mathcal{O}$ such that $p|_o \equiv X$ and $t|_o \equiv t'$ with $t'$ a non-transparent pattern, then $X$ appears only once in $S$, so it appears only once in $S'$ since variables in $S$ and $S'$ are the same. The reasoning is similarly if such a variable $X$ appears in the equation $(h\,\overline{p_n} =^? h\,\overline{t_n})$ since that situation will happen in some equation $(p_i =^? t_i)$.
- (EliminateL) The step is $S \equiv \{X =^? t\} \uplus S'' \Longrightarrow \{X =^? t\} \cup S''[X \mapsto t] \equiv S'$, if $X \in \mathit{fv}(S'') \setminus var(t)$. If $t$ is a non-transparent pattern, then $X$ cannot appear in $S''$ by the transparency of $S$, so this rule cannot be applied. On the other hand, if $t$ is transparent then applying the substitution $[X \mapsto t]$ to $S''$ keeps the left-hand sides transparent. If $X$ appears in the left-hand side of a rule $(p'' =^? t'') \in S''$, we know that if there is a position $o \in \mathcal{O}$ such that $p''|_o \equiv X$ and $t''|_o \equiv t'$ then $t'$ is transparent. Then for all the variables introduced in $p''[X \mapsto t]$ the pattern in the same position in $t''[X \mapsto t]$ will be transparent. If $X$ appears in the right side of some equation, replacing it by $t$ will not generate non-transparent patterns, so there cannot be any equation $(p'' =^? t'') \in S''[X \mapsto t]$ such that $p''|_o \equiv Y$ and $t''|_o \equiv t'$ for some $o \in \mathcal{O}$ and non-transparent pattern $t'$ .
- (EliminateR) The step is $\{p =^? X\} \uplus S'' \Longrightarrow \{p =^? X\} \cup S''[X \mapsto p]$, if $X \in \mathit{fv}(S'') \setminus var(p)$ and $p \notin \mathcal{V}$. The reasoning is the same as the previous case, when $t$ is a transparent pattern.

The proof for any number of steps proceeds trivially by induction on the number of steps.

LEMMA A.5 (Decomposition of patterns). *Let* $h\,\overline{t_n}$ *be a pattern and* $h\,\overline{p_n}$ *be a transparent pattern wrt.* $\mathcal{A}$ *such that* $\mathcal{A} \vdash h\,\overline{t_n} : \tau$ *and* $\mathcal{A} \vdash h\,\overline{p_n} : \tau$*. Then every pair of patterns* $t_i$*,* $p_i$ *verify* $\mathcal{A} \vdash t_i : \tau_i$ *and* $\mathcal{A} \vdash p_i : \tau_i$*, for some* $\tau_i$*.*

**Proof** Since $h\,\overline{p_n}$ is a transparent pattern, $\mathcal{A}(h)$ is n-transparent, so $\mathcal{A}(h) = \forall\overline{\alpha_m}.\overline{\tau'_n} \to \tau'$ such that $var(\overline{\tau'_n}) \subseteq var(\tau')$. Since both patterns have the same type $\tau$, the generic instance $(\mathcal{A}(h) \succ \overline{\tau_n} \to \tau)$ used to derive a type for $h$ in both patterns must be same, forcing the type $\tau_i$ of all the patterns to be the same because $var(\overline{\tau'_n}) \subseteq var(\tau')$.

LEMMA A.6 (Type preservation of $\Longrightarrow$ steps). *Let* $S$ *be a transparent set of equations over patterns and* $\mathcal{A}$ *be a set of assumptions such that for every equation* $(t_1 =^? t'_1) \in S$ *it verifies* $\mathcal{A} \vdash t_1 : \tau$ *and* $\mathcal{A} \vdash t'_1 : \tau$ *for some* $\tau$*. If* $S \Longrightarrow^* S'$ *then for every equation* $(t_2 =^? t'_2) \in S$ *it verifies* $\mathcal{A} \vdash t_2 : \tau$ *and* $\mathcal{A} \vdash t'_2 : \tau$ *for some* $\tau$*.*

**Proof** The proof for one step proceeds by case distinction over the rule of the transition $\Longrightarrow$ applied. All the cases are straightforward with the exception of the (Decompose) case. Since $S$ is transparent, we know that the left-hand side of the equation is transparent, so by Lemma A.5 the step preserves types.

The proof for any number of steps is straightforward using Lemma A.4, as set transparency is preserved.

**Lemma 3.10 (Mgu well-typedness)**
*Let* $\overline{p_n}$ *be fresh linear transparent patterns wrt.* $\mathcal{A}$ *and let* $\overline{t_n}$ *be any patterns such that* $\mathcal{A} \vdash p_i : \tau_i$ *and* $\mathcal{A} \vdash t_i : \tau_i$ *for some type* $\tau_i$*. If* $\theta \equiv mgu(f\,\overline{p_n}, f\,\overline{t_n})$ *then* $wt_\mathcal{A}(\theta)$*.*

**Proof** Easily since $\mathcal{U}(f\,\overline{p_n}, f\,\overline{t_n})$ is the same as the mgu of the set $S \equiv \{p_1 =^? t_1, \ldots, p_n =^? t_n\}$. The set $S$ is transparent—$\overline{p_n}$ are linear and transparent, and no variable in $\overline{p_n}$ appears in appears in $\overline{t_n}$ since they are fresh—so by Lemma A.6 the normal form $S'$ verify that for every equation $(p'_i =^? t'_i)$ both sides have the same type, i.e., $\mathcal{A} \vdash p'_i : \tau_i$ and $\mathcal{A} \vdash t'_i : \tau_i$ for some $\tau_i$. If $S'$ is in solved form then $S'$ has the form $\{\overline{X_n =^? t''_n}\} \cup \{\overline{p''_m =^? Y_m}\}$ so the associated substitution $\theta \equiv [\overline{X_n \mapsto t''_i}, \overline{Y_m \mapsto p''_m}]$ (the obtained mgu) is well-typed because $\mathcal{A} \vdash t''_i : \mathcal{A}(X_i)$ (for $i \in [1..n]$) and $\mathcal{A} \vdash p''_j : \mathcal{A}(Y_j)$ (for $j \in [1..m]$).

### A.3 Proof of Theorem 3.11 :Type preservation of $\rightsquigarrow^{lmgu}$

**Theorem 3.11 (Type preservation of $\rightsquigarrow^{lmgu}$)** *Let $\mathcal{P}$ be a program such that left-hand sides of rules contain only transparent patterns. If $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ and $e \rightsquigarrow_\theta^{lmgu^*} e'$ then $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

**Proof** Straightforward using Theorem 3.8, since under such conditions every $\rightsquigarrow^{lmgu}$-step is a $\rightsquigarrow^{lwt}$-step—trivially if the used rule is (LetIn)–(LetAp), or by Lemma 3.10 if (Narr) is used.

### A.4 Proof of Lemma 4.1: Absence of HO variables

**Lemma 4.1: (Absence of HO variables)** *Let $e$ be an expression such that $wt_{\mathcal{A}}(e)$ and for every $X_i \in fv(e)$, $\mathcal{A}(X_i)$ is not a functional type. Then no step $e \rightsquigarrow_\theta^l e'$ can use (VAct) or (VBind).*

**Proof** If (VAct) is applied then $e$ must contain $X\ \overline{t_k}$, which can only be well-typed if $X$ has a functional assumption in $\mathcal{A}$. On the other hand, if (VBind) is applied then $e$ contains an expression $e' \notin Pat$ such that $e'\theta \in Pat$. It is easy to check that this expression $e'$ must have the form $X\ \overline{t_k}$, so the reasoning is the same as in the previous case.

### A.5 Proof of Lemma 4.6: Absence of unsafe variables

LEMMA A.7 (Decrease of free variables). *If $e \rightsquigarrow^l e'$ using the rules (LetIn), (Bind), (Elim), (Flat) or (LetAp) then $fv(e') \subseteq fv(e)$.*

**Proof** Straightforward.

LEMMA A.8 (Free variables of applied contexts). $fv(\mathcal{C}[e]) = fv(\mathcal{C}) \cup (fv(e) \smallsetminus bv(\mathcal{C}))$

**Proof** Easily by induction on the structure of the context $\mathcal{C}$. The most interesting cases are those involving let-expressions:

- $\mathcal{C} \equiv let\ X = \mathcal{C}'\ in\ e'$.

$$\begin{array}{rll} fv(\mathcal{C}[e]) \equiv & \underline{fv(let\ X = \mathcal{C}'[e]\ in\ e')} & \text{Context application} \\ = & \underline{fv(\mathcal{C}'[e])} \cup (fv(e') \smallsetminus \{X\}) & \text{Definition of } fv \\ = & \underline{(fv(\mathcal{C}')} \cup (fv(e) \smallsetminus bv(\mathcal{C}')))\underline{\cup(fv(e') \smallsetminus \{X\})} & \text{Induction Hypothesis} \\ = & fv(\mathcal{C}) \cup (fv(e) \smallsetminus \underline{bv(\mathcal{C}')}) & \text{Definition of } fv(\mathcal{C}) \\ = & fv(\mathcal{C}) \cup (fv(e) \smallsetminus bv(\mathcal{C})) & \text{Definition of } bv(\mathcal{C}) \end{array}$$

- $\mathcal{C} \equiv let\ X = e'\ in\ \mathcal{C}'$.

$$\begin{array}{rll} fv(\mathcal{C}[e]) \equiv & \underline{fv(let\ X = e'\ in\ \mathcal{C}'[e])} & \text{Context application} \\ = & fv(e') \cup (\underline{fv(\mathcal{C}'[e])} \smallsetminus \{X\}) & \text{Definition of } fv \\ = & fv(e') \cup ((fv(\mathcal{C}') \cup (fv(e) \smallsetminus bv(\mathcal{C}'))) \smallsetminus \{X\}) & \text{Induction Hypothesis} \\ = & \underline{fv(e') \cup (fv(\mathcal{C}') \smallsetminus \{X\})} \cup (fv(e) \smallsetminus (bv(\mathcal{C}') \cup \{X\}) & \text{Set manipulation} \\ = & fv(\mathcal{C}) \cup (fv(e) \smallsetminus \underline{(bv(\mathcal{C}') \cup \{X\})} & \text{Definition of } fv(\mathcal{C}) \\ = & fv(\mathcal{C}) \cup (fv(e) \smallsetminus bv(\mathcal{C})) & \text{Definition of } bv(\mathcal{C}) \end{array}$$

LEMMA A.9. *If $fv(e') \subseteq fv(e)$ then $fv(\mathcal{C}[e']) \subseteq fv(\mathcal{C}[e])$.*

**Proof** Straightforward using the characterization of free variables of an applied context in Lemma A.8.

LEMMA A.10. *Consider the expressions $f\ \overline{t_n}$ and $f\ \overline{p_n}$ and the set of variables $XS \subseteq fv(f\ \overline{t_n})$ such that every variable in $fv(f\ \overline{t_n}) \smallsetminus XS$ is safe wrt. the same $\mathcal{A}$. Consider also a substitution $\theta$ such that $f\ \overline{t_n}\theta \equiv f\ \overline{p_n}\theta$, $dom(\theta) \cap XS = \emptyset$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions over fresh variables used by $\theta$. Then the following conditions hold:*

*a) If $X \in fv(f\ \overline{t_n}) \smallsetminus XS$ then $X\theta$ contain safe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$.*
*b) If $X \in fv(f\ \overline{p_n})$ then every variable $Y \in fv(X\theta)$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$ or $Y \in XS$.*

**Proof**

a) Let $X$ be a variable in $fv(f\ \overline{t_n}) \smallsetminus XS$ with safe type $\mathcal{A}(X) = \tau$. Since $\theta$ is well-typed wrt. $\mathcal{A} \oplus \mathcal{A}'$ and by hypothesis $\mathcal{A}'$ contains only assumptions over fresh variables, then $\mathcal{A} \oplus \mathcal{A}' \vdash X\theta : \tau$, where $\mathcal{A} \oplus \mathcal{A}'(X) = \tau$ remains a safe type wrt. $\mathcal{A} \oplus \mathcal{A}'$. $X\theta$ is a pattern of safe type, so by definition it can only contain safe variables.
b) By *a)* and $dom(\theta) \cap XS = \emptyset$ we know that $f\ \overline{t_n}\theta$ contains variables in $XS$ or safe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$, and since $f\ \overline{t_n}\theta \equiv f\ \overline{p_n}\theta$ then $f\ \overline{p_n}\theta$ contains variables in $XS$ or safe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$ as well.

**Lemma 4.6 (Absence of unsafe variables)** *Let $e$ be an expression not containing unsafe variables wrt. $\mathcal{A}$ and $\mathcal{P}$ be a program such that $wt^r_{\mathcal{A}}(\mathcal{P})$. If $e \rightsquigarrow_\theta^{lwt^*} e'$ then $e'$ does not contain unsafe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

**Proof** We first proceed with the case of one step $e \rightsquigarrow_\theta^{lwt} e'$. The original expression $e$ does not contain any free variable with unsafe type, so it cannot contain free HO variables and by Lemma 4.1 the step $e \rightsquigarrow_\theta^{lwt} e'$ do not use (VBind) or (VAct). Then we proceed by case distinction over the $\rightsquigarrow^{lwt}$ rule used:

- If the rule is (LetIn)–(LetAp) then $\theta \equiv \epsilon$ so $wt_{\mathcal{A}\oplus\mathcal{A}'}(\epsilon)$ for any $\mathcal{A}'$. By hypothesis, every $X \in fv(e)$ is safe wrt. $\mathcal{A}$, so by Lemma A.7 $fv(e') \subseteq fv(e)$ and every $X \in fv(e')$ is safe wrt. $\mathcal{A}$. As $\mathcal{A}'$ is the set of assumptions associated to the step and it contains assumptions over fresh variables, every $X \in fv(e')$ is also safe wrt. $\mathcal{A} \oplus \mathcal{A}'$.
- If the rule is (Narr) then $e \equiv f\ \overline{t_n} \rightsquigarrow^{lwt}_{\theta} r\theta$ for a fresh variant $(f\ \overline{p_n} \to r)$ and $\theta$ such that $f\ \overline{p_n}\theta \equiv f\ \overline{t_n}\theta$. By the hypothesis every variable $X \in fv(f\ \overline{t_n})$ is safe wrt. $\mathcal{A}$, so using Lemma A.10 with $XS = \emptyset$ we obtain that for each $X \in fv(f\ \overline{t_n}) \cup fv(f\ \overline{p_n})$ the pattern $X\theta$ cannot contain any unsafe variable wrt. $\mathcal{A} \oplus \mathcal{A}'$. According to $wt^r_{\mathcal{A}}(\mathcal{P})$ and the definition of the set of assumptions associated to the step, $\mathcal{A}'$ contains ground and safe types wrt. $\mathcal{A}$ for the extra variables in the rule—which are also safe wrt. $\mathcal{A} \oplus \mathcal{A}'$. Any variable $X \in fv(r)$ can be in $fv(f\ \overline{p_n})$ or be an extra varaible. If $X \in fv(f\ \overline{p_n})$ we know that $X\theta$ cannot contain any unsafe variable wrt. $\mathcal{A} \oplus \mathcal{A}'$. On the other hand, if $X \notin fv(f\ \overline{p_n})$ it is an extra varaible, so it is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$ and it is not changed by $\theta$—we assume that $dom(\theta) \subseteq fv(f\ \overline{t_n}) \cup fv(f\ \overline{p_n})$. Therefore every variable in $fv(r\theta)$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$.
- If the rule is (Contx) then $e \equiv \mathcal{C}[e] \rightsquigarrow^{lwt}_{\theta} \mathcal{C}\theta[e']$ for $\mathcal{C} \neq [\,]$, $e \rightsquigarrow^{lwt}_{\theta} e'$ using any rule different from (VAct) or (VBind) and verifying that *i)* $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$ and *ii)* if the rule used is (Narr) with $(f\ \overline{p_n} \to r) \in \mathcal{P}$ then $vran(\theta|_{\smallsetminus var(\overline{p_n})}) \cap bv(\mathcal{C}) = \emptyset$. We distinguish cases on the rule used in the step $e \rightsquigarrow^{lwt}_{\theta} e'$:
  - If the rule used is one of (LetIn)–(LetAp) then $\theta \equiv \epsilon$, so the final expression is $\mathcal{C}[e']$. By Lemma A.7 we know that $fv(e') \subseteq fv(e)$ so by Lemma A.9 $fv(\mathcal{C}[e']) \subseteq fv(\mathcal{C}[e])$. Since we have that every $X \in fv(\mathcal{C}[e])$ is safe wrt. $\mathcal{A}$ from the hypothesis, trivially every $Y \in fv(\mathcal{C}[e'])$ is also safe wrt. $\mathcal{A}$. Finally, as $\mathcal{A}'$ contains assumptions over fresh variables, $Y \in fv(\mathcal{C}[e'])$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$.
  - If the rule used is (Narr) then the step is $\mathcal{C}[f\ \overline{t_n}] \rightsquigarrow^{lwt}_{\theta} \mathcal{C}\theta[r\theta]$ using a fresh variant $(f\ \overline{p_n} \to r) \in \mathcal{P}$ and a unifier $\theta$ such that $wt_{\theta}(\mathcal{A} \oplus \mathcal{A}')$ being $\mathcal{A}'$ the set of assumptions associated to the step—containing ground and safe types for the extra variables of the rule. We assume that $dom(\theta) \subseteq fv(f\ \overline{t_n}) \cup fv(f\ \overline{p_n})$. If we define $XS = bv(\mathcal{C}) \cap fv(f\ \overline{t_n})$ then by Lemma A.10 we know *a)* for every variable $X \in fv(f\ \overline{t_n})$ the pattern $X\theta$ contains only safe variables wrt. $\mathcal{A}\oplus\mathcal{A}'$ and *b)* if $X \in fv(f\ \overline{p_n})$ then every $Y \in fv(X\theta)$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$ or $Y \in XS$. We want to prove that every $Y \in fv(\mathcal{C}\theta[r\theta])$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$. By Lemma A.8 we have that $fv(\mathcal{C}\theta[r\theta]) = fv(\mathcal{C}\theta) \cup (fv(r\theta) \smallsetminus bv(\mathcal{C}\theta))$:
    - $Y \in fv(\mathcal{C}\theta)$. We consider two cases: *1)* $Y \in fv(\mathcal{C})$ but $Y \notin dom(\theta)$. Then $Y \in fv(\mathcal{C}[e])$ (by Lemma A.8), so by hypothesis $Y$ is safe wrt. $\mathcal{A}$, and trivially $Y$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$. *2)* $Y \in fv(Z\theta)$ for some $Z \in fv(\mathcal{C})$. Then $Z \in dom(\theta)$, and as $\overline{p_n}$ has fresh variables then $Z \in fv(f\ \overline{t_n})$. Moreover, since $Z \in fv(\mathcal{C})$ then $Y \notin XS$ because $XS \subseteq bv(\mathcal{C})$. Therefore by *a)* the pattern $X\theta$ contains only safe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$, so $Y$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$.
    - $Y \in fv(r\theta) \smallsetminus bv(\mathcal{C}\theta)$. It is easy to see that $bv(\mathcal{C}) = bv(\mathcal{C}\theta)$ as substitutions does not change bound variables. Since $XS \subseteq bv(\mathcal{C})$ then $Y \notin XS$. Then by *b)* we know that if $Y \in fv(Z\theta)$ for some $Z \in fv(f\ \overline{p_n})$ then $Y$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$. If $Y \in fv(Z\theta)$ for some $Z \notin fv(f\ \overline{p_n})$ then $Z$ is an extra variable and it is not in $dom(\theta)$ because $dom(\theta) \subseteq fv(f\ \overline{t_n}) \cup fv(f\ \overline{p_n})$, so $Y \equiv Z$. Therefore $Y$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$ because $\mathcal{A}'$ contains $\mathcal{A}_f$, where $Y$ has a safe type.

The proof for several steps proceeds by induction on the number $n$ of steps:

BASE CASE: $e_1 \rightsquigarrow^{lwt^0}_{\epsilon} e_1$

Straightforward.

INDUCTIVE STEP: $e_1 \rightsquigarrow^{lwt^{n+1}}_{\theta} e_n \equiv e_1 \rightsquigarrow^{lwt}_{\theta_1} e_2 \rightsquigarrow^{lwt^n}_{\theta'} e_n$

By the proof of one step we have that $e_1 \rightsquigarrow^{lwt}_{\theta_1} e_2$ and every variable in $fv(e_2)$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'_1$, where $\mathcal{A}'_1$ is the set of assumptions associated to the first step. Since $\mathcal{A} \oplus \mathcal{A}'_1$ is $\mathcal{A}$ extended with assumptions over variables, we also have that $wt^r_{\mathcal{A}\oplus\mathcal{A}'_1}(\mathcal{P})$. Therefore by the Induction Hypothesis we have that every variable in $fv(e_n)$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'_1 \oplus \mathcal{A}'$, where $\mathcal{A}'$ is the set of assumptions associated to the reduction $e_2 \rightsquigarrow^{lwt^n}_{\theta'} e_n$.

### A.6 Proof of Theorem 4.7: Completeness of $\rightsquigarrow^{lmgu}$ wrt. $\rightsquigarrow^{lwt}$

**Theorem 4.7 (Completeness of $\rightsquigarrow^{lmgu}$ wrt. $\rightsquigarrow^{lwt}$)** *Let e be an expression not containing unsafe variables wrt. $\mathcal{A}$ and $\mathcal{P}$ be a program such that $wt^r_{\mathcal{A}}(\mathcal{P})$. If $e \rightsquigarrow^{lwt^*}_{\theta} e'$ using mgu's in each step then $e \rightsquigarrow^{lmgu^*}_{\theta} e'$.*

**Proof** By Lemma 4.6 we can assure that no expression involved in the reduction $e \rightsquigarrow^{lwt}_{\theta} e'$ will contain unsafe variables, so by Lemma 4.1 neither (VAct) nor (VBind) are used in the whole reduction. Since $e \rightsquigarrow^{lwt^*}_{\theta} e'$ uses mgu's, by definition $e \rightsquigarrow^{lmgu^*}_{\theta} e'$.

### A.7 Proof of Theorem 5.2: $OIS(\mathcal{P})$ well-typedness

**Theorem 5.2 ($OIS(\mathcal{P}_f)$ well-typedness)** *Let $\mathcal{P}_f$ be a set of program rules for the same function $f$ such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $OIS(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$.*

**Proof** It is easy to check that $wt_{\mathcal{A}\oplus\mathcal{A}'}(f_i\ \overline{t^i_n} \to e^i)$ for each $f_i \in \overline{f_m}$, since $wt_{\mathcal{A}}(f\ \overline{t^i_n} \to e^i)$ and $\mathcal{A}(f) = (\mathcal{A} \oplus \mathcal{A}')(f_i)$. The rule $f\ \overline{X_n} \to f_1\ \overline{X_n}?\ldots\ ?f_m\ \overline{X_n}$ is also well-typed wrt. $\mathcal{A} \oplus \mathcal{A}'$: consider $\mathcal{A}'' \equiv \{\overline{X_n : \tau_n}\}$, where $\mathcal{A}(f) = \forall\overline{\alpha}.\overline{\tau_n} \to \tau$. In this case, $\mathcal{A} \oplus \mathcal{A}' \oplus \mathcal{A}'' \vdash f_i\ \overline{X_n} : \tau$, therefore $\mathcal{A} \oplus \mathcal{A}' \oplus \mathcal{A}'' \vdash f_1\ \overline{X_n}?\ldots\ ?f_m\ \overline{X_n} : \tau$. Since using the $(\Lambda)$ rule it is possible to construct $\mathcal{A}''$, we have the type derivation $\mathcal{A} \oplus \mathcal{A}' \vdash \lambda\overline{X_n}.f_1\ \overline{X_n}?\ldots\ ?f_m\ \overline{X_n} : \overline{\tau_n} \to \tau$. Finally, by Theorem A.1-a it is possible to derive a type $(\overline{\tau_n} \to \tau)[\alpha \mapsto \beta]$ with $\overline{\beta}$ fresh, which is a variant of $\forall\overline{\alpha}.\overline{\tau_n} \to \tau$.

### A.8 Proof of Theorem 5.4: $\mathcal{U}(\mathcal{P})$ well-typedness

**Theorem 5.4 ($\mathcal{U}(\mathcal{P}_f)$ well-typedness)** *Let $\mathcal{P}_f$ be a set of program rules for the same overlapping inductive sequential function $f$ such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $\mathcal{U}(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$..*

**Proof** *(Sketch)* We will see that the new rules $\mathcal{P}''$ added in each step are well-typed wrt. $\mathcal{A}\oplus\mathcal{A}''$, where $\mathcal{A}''$ are the assumptions added in the step. Consider the rule and assumption added for $\mathcal{P}_i$: $f\ \overline{X_j}\ (c_i\ \overline{Y_{k_i}})\ \overline{Z_l} \to f_{(c_i,o)}\ \overline{X_j}\ \overline{Y_{k_i}}\ \overline{Z_l}$ and $\mathcal{A}''(f_{(c_i,o)}) = \forall\overline{\alpha}.\overline{\tau_j} \to \overline{\tau'_{k_i}} \to \overline{\tau_l} \to \tau$, where $\mathcal{A}(f) = \forall\overline{\alpha}.\overline{\tau_j} \to \tau' \to \overline{\tau_l} \to \tau$ and $\mathcal{A} \oplus \{\overline{Y_{k_i} : \tau'_{k_i}}\} \vdash c_i\ \overline{Y_{k_i}} : \tau'$ by the definition of $\mathcal{U}$ (Definition 5.3). It is clear that

$$\mathcal{A} \oplus \mathcal{A}'' \oplus \{\overline{X_j : \tau_j}, \overline{Y_{k_i} : \tau'_{k_i}}, \overline{Z_l : \tau_l}\} \vdash f_{(c_i,o)}\ \overline{X_j}\ \overline{Y_{k_i}}\ \overline{Z_l} : \tau$$

and

$$\mathcal{A} \oplus \mathcal{A}'' \oplus \{\overline{X_j : \tau_j}, \overline{Y_{k_i} : \tau'_{k_i}}\} \vdash c_i\ \overline{Y_{k_i}} : \tau'$$

Therefore we can build the type derivation for the $\lambda$-abstraction

$$\mathcal{A} \oplus \mathcal{A}'' \vdash \lambda\overline{X_n}.\lambda c_i\ \overline{Y_{k_i}}.\lambda\overline{Z_l}.f_{(c_i,o)}\ \overline{X_j}\ \overline{Y_{k_i}}\ \overline{Z_l} : \overline{\tau_j} \to \tau' \to \overline{\tau_l} \to \tau$$

Finally, by Theorem A.1-a it is possible to derive any variant of $\mathcal{A}(f)$ for this $\lambda$-abstraction by using $[\overline{\alpha \mapsto \beta}]$ with $\overline{\beta}$ fresh, so the rule is well-typed. Notice that the recursive call of the transformation can introduce assumptions for new functions, but the previous derivation remains valid by by Theorem A.1-b, since these new functions cannot appear in the expression. Therefore, the rule is well-typed wrt. the final set of assumptions $\mathcal{A}'$ returned by the transformation.