Typing as Functional-Logic Evaluation

Francisco J. López-Fraguas  
Dpto. de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain  
fraguas@sip.ucm.es  

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

Abstract

We present a transformational approach to type inference for functional logic programs. More concretely, we give a broad set of examples showing how, given a functional logic program P, we can synthesize a remarkably simple and natural functional logic program P’ such that the evaluation of expressions with respect to P’ corresponds to typing the expressions in the original P. We start developing those ideas for the case of type inference with standard Hindley-Milner types, and after that we consider some variations, like local definitions with different degrees of polymorphism, existential types and type checking in the presence of polymorphic recursion. For the basic case of Hindley-Milner types we provide also a formalization of the transformation and proofs of its correctness. Besides its potential applicability to the implementation of different type systems, or to the educational use of the synthesized typing programs to explain different type inference/checking processes, the paper demonstrates vividly the expressive power of functional logic languages, as well as some of their limitations for metaprogramming purposes, that we have overcome by providing a suitable set of metalevel functions to inspect, classify and manipulate expressions according to their structure, similar to well known Prolog metapredicates for such purposes.

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

General Terms Design, Languages

Keywords Type inference, type checking, functional logic programming

1. Introduction

Declarative languages are good candidates to create executable specifications of typing procedures for Hindley-Milner-like type systems, which are common in many programming languages. Functional programming based specifications exist since long ago (see e.g. [24], Chapter 9); a particularity is that they must encode and handle explicitly substitutions and unification, two core ingredients of Hindley-Milner type inference. In this sense, logic programming based specifications have an advantage, since handling of substitutions and unification are built into the system. This is even more apparent with a constraint-based view of type inference [25, 26], that reached a great generality with the HM(X) scheme [23, 27]. The starting working thesis of this paper is that functional logic programming [2, 10, 11], as an integrated paradigm taking the best features of functional and logic languages, could be an interesting alternative.

The paper develops that thesis to a certain extent: we show how functional logic programs can be used to obtain remarkably simple and clear specifications of Hindley-Milner-type type inference and checking procedures by following a transformational approach: given a functional (or functional-logic) program written, as usual, as a collection of datatype and function definitions, we synthesize an associated typing program, which is a functional logic program such that evaluating an expression in it returns the type of the corresponding expression in the original program (or finitely fails, in a sense similar to Prolog, if the expression was ill-typed).

We present things gradually: we start with some basic examples of inference of Hindley-Milner polymorphic types, for which we formalize the transformation and prove some technical results showing that the transformed typing program reflects adequately what happens at the level of type derivations for the original one. After that, we consider in a example-driven manner a small set of variations and extensions: local definitions with different degrees of polymorphism; existential types; type errors; type checking and inference in presence of polymorphic recursion.

We will see in the paper the convenience of extending current functional-logic systems with some meta-programming facilities that are typical for logic systems like Prolog, mostly related to inspection of terms: check if a term is a variable, check if a term contains another of a given shape, subsumption checks, etc. They are not considered in existing systems because they must be seen as “impure” features not expressible within the standard semantic frameworks for functional logic programming.

The rest of the paper has a simple organization: after some preliminaries describing functional logic programs, we present in Section 3 the basic case of Hindley-Milner types as well as the formalization of the transformation and associated results. Sections 4 and 5 contain the announced extensions and some conclusions, respectively. The code of the examples contained in this paper can be found at http://gpd.sip.ucm.es/trac/gpd/wiki/TypingFLComputation

2. Preliminaries

Along the examples in the paper, both the source and the typing programs are written in the concrete syntax of the functional logic language Toy [18], but should in principle be easily adapted to other FLP languages like Curry [12]. A Toy program is composed of data type declarations, type aliases, infix operator declarations, function type declarations and defining rules for function symbols. Toy’s syntax is essentially Haskell’s, but with Prolog’s rule for capitalization: variables begin with upper-case letters, identifiers...
for types, constructors and functions use lower-case. Each defining rule for a function \( f \) takes the form
\[
f \ t_1 \ldots t_n = e \iff e_1 = e'_1, \ldots, e_k = e'_k
\]
where \( (t_1, \ldots, t_n) \) forms a tuple of linear constructor terms, and \( e, e_1, e'_1 \) are expressions. Rules have a conditional reading: \( f \) \( t_1 \ldots t_n \) can be reduced to \( e \) if all the conditions \( e_1 = e'_1, \ldots, e_k = e'_k \) are satisfied. The condition part is omitted if \( k = 0 \). The symbol \( \iff \) stands for strict equality: \( e = e' \) means that \( e \) and \( e' \) can be reduced to the same constructor term.

Functions in Toy can be non-deterministic, either because of overlapping rules, as in
\[
\begin{align*}
\text{choice } X \ Y &= X \\
\text{choice } X \ Y &= Y
\end{align*}
\]

or because of the occurrence of extra variables in right-hand sides, as in
\[
\text{sublist } Xs = Ys \iff Us ++ Ys ++ Zs = Xs
\]
where ++ is list concatenation. Overlapping rules are not used in this paper, but extra variables are indeed instrumental in it.

Computing in Toy means solving goals, which take the form \( e_1 = e'_1, \ldots, e_k = e'_k \) giving as its result a substitution for the variables in the goal making it true. Evaluation of expressions (required for solving the conditions) is done by needed narrowing \([3]\), a variant of lazy narrowing which uses definitional trees \([1]\) to guide unification with patterns in left-hand sides of rules. There exists a formal semantics for Toy programs, which is shortly recalled in Section \([4]\).

3. Hindley-Milner Typing as Functional-Logic Evaluation

In this section we will present how to translate an FL program into another FL program such that the evaluation of expressions returns their types, considering a type system similar to Hindley-Milner \([3]\). Therefore, first of all we need to define a data type of types to represent returned types. This data type must contain a constructor to represent the functional type \((\to)\), as well as constructors to represent every type declared in the original program. For example, consider an original program containing the following data types\([1]\):

\[
\begin{align*}
data boolean &= \text{true} | \text{false} \\
data nat &= \text{z} | \text{s} \text{nat} \\
data list A &= \text{nil} | \text{cons} A \text{ (list} A) \\
data pair A B &= \text{cpair} A B
\end{align*}
\]

The resulting data type of types contains the “arrow” constructor for functional types and constructors for booleans, natural numbers, polymorphic lists and pairs:

\[
\begin{align*}
\text{infixr } 60 \to & \\
data types &= \text{types} \to \text{types} | \text{t_nat} | \text{t_boolean} \\
&\quad | \text{t_list types} | \text{t_pair types types}
\end{align*}
\]

As a convention, we will append the prefix \( \text{t}_\cdot \) to every symbol in the translated program which comes from the original program.

The translation of data types is easy since each data type declared as data \( A_1 \ldots A_n = (...) \) must generate a new constructor \( A_1 \ldots A_n \) in the resulting types data type. Regarding concrete data constructors defined in the original program, they produce new function symbols in the translated program. Each of these function symbols has one rule, which returns the representation of the type of the data constructor in the original program.

It is important to note that type variables in the type of the data constructor—which are implicitly universally quantified—are translated into extra variables in right-hand sides of rules. For example the data constructor \( \text{cpair} \) with type \( \forall A, B \to \beta \to \text{pair} \to \alpha \beta \) generates the rule \( \text{t_cpair} = A :\to B :\to \text{t_pair} A B \), where both \( A \) and \( B \) are extra variables in the rule (therefore, this generated rule would not be legal in a functional language). These extra variables may be unified to concrete types during evaluation—by narrowing—depending on the context where the symbol appears, thereby allowing that the type could be used with different generic instances in different places. Considering the previous data declarations, the generated functions are the following:

\[
\begin{align*}
t_{\text{true}}, t_{\text{false}}, t_{\text{z}}, t_{\text{s}}, t_{\text{nil}} &\quad \text{t_cons, t_cpair} \quad \text{types} \\
t_{\text{t_true}} &= t_{\text{t_boolean}} \\
t_{\text{t_false}} &= t_{\text{t_boolean}} \\
t_{\text{t_z}} &= t_{\text{t_nat}} \\
t_{\text{t_s}} &= t_{\text{t_list}} A \quad \% \text{extra variable } A \\
t_{\text{t_nil}} &= t_{\text{t_list}} A \quad \% \text{extra variable } A \\
t_{\text{t_cons}} &= A :\to t_{\text{t_list}} A \quad :\to t_{\text{t_list}} A \\
t_{\text{t_cpair}} &= A :\to B :\to \text{t_pair} A B \quad \% \forall A, B \text{ extra vars}
\end{align*}
\]

We need a way of expressing application of expressions, since in the transformed program expressions evaluate to their types, which are represented by non-functional values. For that we will use the left-associative infix operator \( \&\& \), whose program rule infers the type of an application according to the usual typing rule: \( e_1 e_2 \) has type \( \tau \) if \( e_1 \) has type \( \tau_1 \to \tau \) and \( e_2 \) has type \( \tau_1 \). Since functions in the original program can have several rules, we also need an operator \( \setminus \) to unify the types obtained for the different rules.

\[
\begin{align*}
\text{infixl } 60 \&\& &\% \text{ application} \\
\text{infixr } 50 \setminus &\% \text{ type unification}
\end{align*}
\]

\[
\begin{align*}
(\&\&), (\setminus) \quad :\to \quad \text{types} \to \text{types} \to \text{types} \\
(\text{t_true} :\to \text{t_f} ) \&\& \text{t_f} = \text{t_f} \\
\% \text{ Desugared: } \text{t_true} :\to \text{t_f} \quad \text{t_f} = \text{t_f} \\
\text{T} \setminus \text{T} = \text{T} &\% \text{ Desugared: } \text{T} / \text{T} \quad \text{T} = \text{T}
\end{align*}
\]

With the previous functions, we can now translate user functions to obtain versions that are evaluated to their types. Conceptually, each rule is translated into a functional type that is composed by the translation of the patterns of the original rule, followed by the translation of the right-hand side of the original rule: \( f t_1 \ldots t_n = e \) is translated into \( t_f = t'_1 :\to \ldots :\to t'_n :\to e' \). In the translated
expressions \( t'_i \) and \( e' \), the symbols from the original program are replaced by their translated version (with prefix \( t_\text{tr} \)) and application is replaced by the infix operator \( \text{foo} \). Considering \( \text{isEmpty} \) in Figure 1, the translation would be \( t_\text{isEmpty} = t_\text{nil} \rightarrow t_\text{tr} \).

To obtain the type of \( \text{isEmpty} \), we only have to evaluate the translated function in the Toy interpreter, using the translated program.

Toy> \( t_\text{isEmpty} = T \)
\{ \( T \rightarrow (t_\text{list } _A) \rightarrow t_\text{true} \) \}

Please do not confuse the first arrow \( \rightarrow \) expressing the binding of the variable \( T \) in the Toy computation with the arrow \( \rightarrow \) in the computed answer representing a functional type. Using the transformed program, we can infer the type not only of function symbols, but of any expression, like \( \text{isEmpty} \) nil:

Toy> \( t_\text{isEmpty} \text{ nil} = T \)
\{ \( T \rightarrow t_\text{false} \) \}

As an example of a function with several rules, consider the choice function in Figure 1 that non-deterministically returns one of its arguments. In this case the function has two rules, so its type must be the most general type that can be inferred for them. In order to obtain that type, we use the \( \backslash / \) operator, which unifies the types inferred for both rules:

\[
\begin{align*}
\text{t-choice} & = (X :\rightarrow Y :\rightarrow X) \backslash / (X1 :\rightarrow Y1 :\rightarrow Y1) \\
\text{Toy> t-choice} & = T \\
& \{ T \rightarrow \_A :\rightarrow \_A \rightarrow \_A \}
\end{align*}
\]

As expected, the obtained type for choice is \( \alpha \rightarrow \alpha \rightarrow \alpha \). Notice that we have used a variant of the second rule, renaming its variables to \( X1 \) and \( Y1 \). In this case using a variant does not change the obtained type, however in general it is essential to guarantee that the different rules have independent variables. Otherwise, the type inference can fail or an incorrect type can be inferred, because a variable with the same name in different rules is forced to be the same.

Up to now, we have considered only non-recursive functions, however the treatment of recursion would only add a slight modification. For example, consider the function map in Figure 1. In order to infer a correct type for a recursive function, we need to replace each occurrence of the recursive function by the same free variable, which denotes the returned type of the translated function. With this replacement, the type inference will take into account the type inferred for each rule as well as the type “forced” by each call. This is similar to usual inference, which starts with a type variable in the type environment for the function, and updates that variable by unification with the types inferred for the rules or by the types of the arguments in each call. In the proposed translation, the rule returns the free variable, which is forced to be the type inferred for the rules—using an equality condition—where each occurrence of the recursive function is replaced by the same extra variable:

\[
\begin{align*}
\text{t-map} & = A \leftarrow A \leftarrow (F :\rightarrow t_\text{nil} \backslash / F1 :\rightarrow t_\text{cons} \quad X \quad Xs :\rightarrow t_\text{cons} \quad (F100X) \quad (A \quad Xs) ) \\
\text{Toy> t-map} & = T \\
& \{ T \rightarrow (\_A :\rightarrow \_B) :\rightarrow (t_\text{list } \_A) :\rightarrow (t_\text{list } \_B) \}
\end{align*}
\]

As the previous example shows, the rule returns the extra variable \( A \), which is equal to the translation of the original rules of the function. Each occurrence of map in the right-hand side is replaced by \( A \), and as before variants are used for those variables appearing in previous rules—as \( F1 \) in the second rule.

Variable \( X, Y, Z \ldots \)
Function \( f \)
Data constructor \( c \)
Type constructor \( C \)
Program \( P := \text{data}^* \text{ fdecl}\)
Datatype \( \text{data} ::= \text{data } c \quad X_n = c_1 \quad \tau_{\alpha_1} \quad \ldots \quad c_m \quad \tau_{\alpha_m} \)
Type \( \tau := X \quad C \quad \tau_1 \ldots \tau_n \quad \tau_1 \rightarrow \tau_2 \)
Function decl. \( \text{fdecl} ::= R_1^l \ldots R_k^l \) (for the same function \( f \))
Program rule \( R_1^l ::= f_1 \ldots \tau_n \quad e \quad (\text{linear}) \)
Pattern \( t ::= X \quad | \quad C \quad | \quad f \quad | \quad c \quad t_1 \ldots \tau_n \) (\( n \leq \alpha(c) \))
Expression \( e ::= X \quad | \quad C \quad | \quad f \quad | \quad e \quad e \)

Figure 2. Syntax of source programs

To finish this section about Hindley-Milner type inference, we will consider mutually dependent functions. A clear example of mutually dependent functions are \( \text{even} \) and \( \text{odd} \) in Figure 1. Contrary to previous examples, type inference for sets of mutually dependent functions must be done at the same time because the type of one function depends on the type of the rest. To obtain that behavior we generate an auxiliary function that returns a tuple of extra variables, which are forced to be the types of the different functions by the condition—as before, we replace every recursive function call for the corresponding variable, and use variants of the rules. The translation for \( \text{even} \) and \( \text{odd} \) is as follows:

\[
\begin{align*}
\text{odd-even} & = (A,B) \leftarrow A \leftarrow (t_\text{fl} \backslash / t_\text{st} \quad X \quad X :\rightarrow B \quad X, B \leftarrow (t_\text{tr} \quad \backslash / t_\text{st} \quad X \quad X1 :\rightarrow A \quad X1) \\
\text{t-odd} & = A \quad \text{where } (A,B) = \text{odd-even} \\
\text{t-even} & = B \quad \text{where } (A,B) = \text{odd-even} \\
\text{Toy> t-even} & = T \\
& \{ T \rightarrow t_\text{nat} :\rightarrow t_\text{false} \} \\
\text{Toy> t-odd} & = T \\
& \{ T \rightarrow t_\text{nat} :\rightarrow t_\text{true} \}
\end{align*}
\]

The rule that returns the type of each of the mutually dependent functions simply uses the auxiliary function and projects the corresponding component of the tuple.

3.1 Formalization of the translation and properties

In the previous section we introduced by example the most important points of the translation. Now we formalize it and provide some results showing its adequacy w.r.t. the type system.

Figure 2 contains the concrete syntax considered for source programs. Notice that this syntax is the same as the one informally presented in Section 1 but omitting some syntactic sugar constructions like type alias or infix operators declarations for the sake of clarity. We have also omitted function type declarations because our approach does not specify type checking but complete type inference (more information about extensions to perform type checking can be found in Section 4.1). In the sequel we will use \( \tau_n \) to denote a sequence \( o_1 \ldots o_n \) of \( n \) syntactic elements \( o \) (we will omit the subscript if the number of elements is not important). A program \( P \) consists of zero or more data declarations followed by zero or more function declarations. Data declarations have the standard syntax in functional or functional-logic languages. However, as existing types \( [13][21] \) are not considered in our type system (but see Section 4.2), for each constructor \( c \) defined with \( \text{data } c \quad X_n = \ldots \quad c_i \quad \tau_{\alpha_i} \quad \ldots \) we restrict the variables in \( \tau_{\alpha_i} \) to be a subset
of \(\{X_n\}\). Function declarations are sets of program rules for the same function symbol. For each program rule \(t_1 = t_n = e\), \(\tau_n\) must be linear, i.e., there is no repetition of variables. We will consider that program rules have fresh variables, so no variable can appear in different rules. Notice that program rules can have extra variables—variables in \(e\) that do not occur in the arguments \(\tau_n\)—as mentioned in Section 2.

The arity of a constructor or function symbol is the number of arguments in its definition. Therefore, we will say that the data constructor \(c_i\) defined with \(\text{data} C X_n = \ldots \mid c_i; \tau_{m_i} \ldots\) has arity \(k_i = \text{arity}(c_i) = k_i\)—and the function symbol \(f\) defined with a rule \(\leq f t_1 = t_n = e\) has arity \(\text{arity}(f) = |f|\).

The syntax of the programs obtained using the translation is the same as the syntax in Figure 2 with only one difference: rules in transformed program can contain conditions. As explained in Section 2 these conditions are strict equalities between expressions, so the syntax of a program rule is

\[
\leq f t_1 = \ldots t_n = e \iff e_1 = \ldots e_n = e',
\]

Conditions are used intensively in the translated program, but we have excluded them from source programs for the sake of clarity. This exclusion must not be understood as a limitation, since conditions can be easily encoded as if/then expressions, which from the point of view of types are indistinguishable.

We will use square brackets \([\ldots]_p\) to denote the translation, where the subscript \(p\) will distinguish the concrete translation used: programs, data declarations, expressions, types, etc. The translation of a program is simply the translation of its data declarations and function rules together with the rules for \(\emptyset\) and \(\wedge\). Notice that in the following translation rules we only focus on the new function rules produced, so the generation of the datatype types and its data constructors (which is straightforward) is omitted.

**Definition 1** (Translation of programs).

\[
\begin{align*}
\text{data}_1 &\ldots \text{data}_n \mid \text{fdecl}_m \mid \tau \mid \tau_n \\
\text{data}_1 \mid \ldots \mid \text{data}_n \mid \text{data} &\mid \text{fdecl}_m \mid \tau \mid \tau_n
\end{align*}
\]

(T :-> \(T'\)) \(\emptyset\) \(\emptyset\) \(T'\) \(\iff T == T'\),

To translate the section of data declaration, we translate its datatype declarations. For each constructor in a datatype declaration we generate a function rule that returns its type.

**Definition 2** (Translation of data declarations).

\[
\begin{align*}
\text{data} C X_n = c_1; &\ldots; c_m &\text{data} = \\
\text{t}_c c_1 &\mid [\tau_1]_\text{type} : \ldots : \text{t}_c C X_1 \ldots X_n &\text{...} \\
\text{t}_c c_m &\mid [\tau_m]_\text{type} : \ldots : \text{t}_c C X_1 \ldots X_n
\end{align*}
\]

**Types**

\[
\begin{align*}
[X]_\text{type} &\equiv X \\
[C \tau_1 &\ldots \tau_n]_\text{type} &\equiv \text{t}_c C [\tau_1]_\text{type} \ldots [\tau_n]_\text{type} &\text{...} \\
[\tau_1 &\rightarrow \tau_2] &\equiv [\tau_1]_\text{type} : \ldots : [\tau_2]_\text{type}
\end{align*}
\]

To translate the section of function declarations, we need to translate each program rule. However, we first need to create sets of rules for mutually dependent functions, as each set is translated as a whole. Notice that we need to apply a replacement \(\omega\) when dealing with sets of mutually dependent function rules. This replacement

---

3 All the rules for the same function symbol \(\tau\) must have the same number of arguments.

maps translated function symbols \(\tau_{f_i}\) to variables; this is needed to support recursion, as explained before. A difference of \([\ldots]_{\text{fdecl}}\) with respect to the translation informally presented is that it treats the sets of rules in a homogeneous way, without distinguishing between sets of program rules that contain mutually dependent functions and sets containing rules for only one function. Then, even for these latter sets, it will create the “auxiliary” function \(f\) that will return the “tuple” of types and the projecting function \(\tau_{f_i}\). In practice it could be optimized to generate the same function explained informally, but we have considered that homogeneity in the formalization of the translation provides clarity.

**Definition 3** (Translation of program rules).

**Expressions**

\[
\begin{align*}
[X]_\text{exp} &\equiv X \\
[c]_\text{exp} &\equiv \text{t}_c c \\
f[\ldots]_\text{exp} &\equiv \text{t}_f \ldots f \\
c_1 c_2 &\equiv \text{c}_1 \text{exp} \emptyset \emptyset \text{c}_2 \text{exp}
\end{align*}
\]

**Function declarations**

\[
\begin{align*}
\text{fdecl}_1 &\ldots \text{fdecl}_n \mid \text{fdecl}_m = \bigcup_{m=1}^n \text{FD}_m \mid \text{fdecl}_n &\equiv \bigcup_{m=1}^n \text{FD}_m \mid \text{fdecl}_n
\end{align*}
\]

where the function declarations \(\text{fdecl}_1 \ldots \text{fdecl}_n\) have been split into \(k\) sets of function declarations for mutually dependent functions \(\text{FD}_i\).

**Mutually dependent function declarations**

\[
\begin{align*}
\text{fdecl}_1 &\ldots \text{fdecl}_n &\equiv &\text{fdecl}_1 \ldots \text{fdecl}_n &\equiv &\text{fdecl}_1 \ldots \text{fdecl}_n \\
f_1 &\ldots f_n &\equiv &\text{f}_1 \ldots \text{f}_n &\equiv &\text{f}_1 \ldots \text{f}_n
\end{align*}
\]

where \(\text{fdecl}_i\) contains \(k_i\) program rules \(R_i^1 \ldots R_i^n\) and the replacement \(\omega\) is defined as \(\omega \equiv [\text{t}_f \text{f}_1 \rightarrow \text{a}_1, \ldots, \text{t}_f \text{f}_n \rightarrow \text{a}_n]\)

**Program rules**

\[
\begin{align*}
[f]_\text{rule} &\equiv \text{t}_f f &\rightarrow &\text{t}_f [\ldots]_\text{exp} &\rightarrow &\text{t}_f [\ldots]_\text{exp}
\end{align*}
\]

To express the adequacy of the translation, we need to establish the concrete type system and the semantics used to evaluate expressions in the translation program. Regarding the type system, we will use the one proposed in [10], whose rules can be found in Figure 3.

This recently proposed type system guarantees type preservation under narrowing derivations, providing support for extra variables...
in program rules. Type derivations have the shape $A |- e : \tau$, meaning that $\tau$ can be derived for the expression $e$ under the set of assumptions $A$. A set of assumptions $A$ is a mapping from symbols $s$ (variables, constructors or function symbols) to type-schemes $\sigma (\{\tau_n, \ldots, \tau_i\})$, where type-schemes are simple types with some variables universally quantified (defined as $\sigma := \tau \lor \{X, \sigma\}$). We will use the notation $A(s)$ to express the type-scheme related to the symbol $s$ in $A$.

The typing rules in Figure 3 are very similar to those in the Damas-Milner type system [4, 5]. For symbols, the (ID) rule derives an instance of the type stored in the set of assumptions $A$. This is obtained using the instantiation operator $\times$. This operator is defined as: $\sigma \times \tau$ if $\sigma \equiv \forall X.\tau'$ and $\tau \equiv \tau'[X_n \mapsto \tau_n]$ for some types $\tau_n$, i.e., if $\tau$ results from replacing the universally quantified variables $\sigma$ by some types. The (APP) rule is standard for deriving types of applications, and ($\lambda$) derives a type for a $\lambda$-abstraction. Although the latter kind of expressions cannot appear in programs, they are important to define the notion of well-typed rules and well-typed programs—see [6] for details. Notice that since ($\lambda$) guesses types for all the variables in the $\lambda$-abstraction (the variables $\forall t$ in the pattern $t$ as well as the free variables $fv(\lambda t.e)$ in the whole expression) extra variables are supported by the type system.

Since the typing rules in Figure 3 only cover expressions, we need a notion to establish when a program rule and a whole program are well-typed [6]:

**Definition 4 (Well-typed program).** A program rule $f \vdash \tau_n = e$ is well-typed w.r.t. a set of assumptions $A$ iff $A \vdash \lambda t_1 \ldots \lambda t_n.e: \tau$ and $\forall t \vdash \tau[fv(e)] = \forall t \vdash \tau[fv(e)] = \forall t \vdash \tau[fv(e)]$ for some fresh variables $\tau_n$. A program $P$ is well-typed w.r.t. $A$—written $\text{wt}(A, P)$—iff all its rules are well-typed w.r.t. $A$.

Regarding the semantics, we will use the HO-CRWL framework [10]. It is a higher-order extension of CRWL [9], a well-established semantics in the FLP community [11], supporting the main features of FLP like call-time choice and non-determinism. HO-CRWL derivations, defined by the proof calculus of Figure 4, have the form $P \vdash_{\text{CRWL}} e \rightarrow t$, meaning that the (partial) pattern $e$ approximates a value of the evaluation of the expression $e$ under the set of program rules $P$ (note that program rules, expressions and patterns are defined as in Figure 3). When the set of program rules $P$ is obvious from the context, we will usually omit it from derivations and simplify them to $e \rightarrow t$. The proof calculus includes a rule (J) to resolve strict equalities $e_1 \equiv e_2$ (also known as joinability statements) that appear in the conditions of the rules. To deal with non-strictness, HO-CRWL considers a special semantic element $\bot$ in patterns. Therefore, we will make a distinction between those patterns containing $\bot$ (partial patterns, Pat$_\bot$) and those that do not contain $\bot$ (total patterns, Pat$_t$). The rules of HO-CRWL also use data substitutions $\{X_n \mapsto \tau_n\} \equiv \theta \in \text{PSubs}$, i.e., mappings from variables to patterns. Similarly to patterns, data substitutions can be total ($\text{PSubs}$), if they map variables to total patterns, or partial ($\text{PSubs}_\bot$), if they map variables to partial patterns. The proof calculus should not be understood as an operational procedure, but rather as an abstract suitable description of the expected results. Computations are performed in practice by some kind of lazy demand-driven narrowing, modeled formally by different notions of term or graph rewriting and narrowing [3, 6, 7, 19] that are equivalent in some technical sense to HO-CRWL derivations (see [7, 19]).

Now that we have a type system and a semantic setting, we can state the adequacy results of the translation. The first result relates type derivations and the evaluation of translated expressions using translated set of assumptions. The translation of a set of assumptions simply generates a program rule for each constructor and function symbol that returns the translation of its type, as well as the rule to evaluate the application of functional types:

**Definition 5 (Translation of closed sets of assumptions).**

\[ \left[ \{ \tau_n : \forall X.\tau_n \} \}_{\text{assump}} \equiv \begin{cases} \tau_n & \text{if } \tau_n \text{ is a variable} \\ \tau & \text{if } \tau \text{ is a type-scheme} \\ \tau \times \sigma & \text{if } \tau \times \sigma \text{ is a type-scheme} \\ \forall t \vdash \tau[fv(e)] & \text{if } \forall t \vdash \tau[fv(e)] \text{ is a type-scheme} \\ \tau_1 \rightarrow \tau_2 & \text{if } \tau_1 \rightarrow \tau_2 \text{ is a type-scheme} \\ \tau_1 \times \tau_2 & \text{if } \tau_1 \times \tau_2 \text{ is a type-scheme} \end{cases} \]

**Figure 4. HO-CRWL rules**

According to this translation, universally quantified variables in a type-scheme are translated into extra variables of a program rule, so they can be instantiated to any type during evaluation. For example the assumption $\{ id : \forall X.X \rightarrow X \}$ is translated into $\tau_1 \vdash id = X \rightarrow X$, so we can derive any instance type like $\tau_1 \vdash t \text{ bool} \rightarrow \tau_1 \vdash t \text{ bool} \rightarrow \tau_1 \vdash (\text{t_list} \text{ t_nat}) \rightarrow (\text{t_list} \text{ t_nat})$.

Notice that this translation only works for closed sets of assumptions, i.e., sets of assumptions whose type-schemes do not have free variables, because it treats all variables as universally quantified. This is not a flaw in our results, as closed sets of assumptions are the only sensible sets of assumptions that can be related for programs (non-closed sets of assumptions are also useful, but they only appear as intermediate states when inferring types). Notice also that the translation of a set of assumptions containing assumptions for variables $X : \forall X.\tau$ is undefined.

Using the translation of sets of assumptions in Definition 5, we can state the equivalence between type derivations and CRWL-derivations of translated closed expressions (expressions not containing variables) using the translated set of assumptions:

**Theorem 1.** Consider a closed set of assumptions $A$ for constructor and function symbols and a closed expression $e$. Then $A \vdash e : \tau$ \iff $[A]_{\text{assump}} \vdash \text{CRWL} \vdash [e]_{\text{exp}} \rightarrow [\tau]_{\text{type}}$.

This result does not consider translated programs but translated sets of assumptions. It is important, however, because it shows that, if the same set of assumptions is considered, translated expressions are evaluated to the same types that can be derived using the type system, and vice versa.

To state the next result relating evaluations using the translated set of assumptions $[A]$ and the translated program $[P]$, we will need to define data-coherent sets of assumptions w.r.t. programs:

**Definition 6 (Data coherence).** A set of assumptions $A$ is data-coherent w.r.t. a program $P$ if for each data constructor $c_i$ defined in $P$ as:

| B | e \rightarrow \bot |
| RR | X \rightarrow X |
| DC | e_1 \rightarrow t_1 \ldots e_n \rightarrow t_m | \text{if } h_1 \ldots e_m \rightarrow t_1 \ldots t_m \in \text{Pat}_\bot, m \geq 0 |
| OR | e_1 \rightarrow t_1 \ldots e_n \rightarrow t_0 \theta | \text{if } m \geq 0, (f_1 \ldots e_n = e') \in P, \theta \in \text{PSubs}_\bot |
| J | e_1 \rightarrow t | e_2 \rightarrow t | \text{if } \theta \in \text{PSubs}_\bot |

Now $P \vdash c_i : \forall X_n.\tau_n \rightarrow \ldots \rightarrow \forall t_{i_1} \rightarrow \forall \tau_n$.
Data-coherence is vital to relate evaluation using $[A]$ and $[P]$, since it guarantees that $A$ contains the assumptions for constructors that reflect the data declarations in the program. For example, if a program $[P]$ contains the data declaration

```
data boolean = tr | fl
```

but the non-data-coherent set of assumptions contains

```
\{ tr : nat, fl : nat \}
```

then it is clear that evaluation under $[A]$ and $[P]$ will be unrelated because the only value of $[tr]$ is $t, tr$ under $[A]$ is $t, nat$ ($[A]$ $\vdash_{\text{CRWL}} [t] \rightarrow t, nat$), whereas the only evaluation using $[P]$ is $[P] \vdash_{\text{CRWL}} [t] \rightarrow t, boolean$—see the translation rules for data declarations in Figure 2.

Using the previous definition we can state that for well-typed programs and data-coherent sets of assumptions, every derived value for the translation of an expression $[e]$ under $[A]$ will also be a value under $[P]$:

**Theorem 2.** Consider a program $P$ and a set of assumptions $A$ such that $P$ is well-typed w.r.t. $A$—with $A(P)$—and $A$ is data-coherent w.r.t. $P$. Then $\forall e \in \text{Exp} \ni t \in \text{Pat}. [A]_{\text{assump}} \vdash_{\text{CRWL}} [e]_{\text{exp}} \rightarrow t \Rightarrow [P]_{\text{prop}} \vdash_{\text{CRWL}} [e]_{\text{exp}} \rightarrow t$.

Notice that the previous theorem holds for any total pattern $t \in \text{Pat}$, so in particular it will also hold for translations of types $[\tau]$. A straightforward consequence of the two previous theorems is the next corollary, relating type derivations and evaluation under translated programs:

**Corollary 1 (Adequacy).** Consider a closed expression $e$, a program $P$ and a closed set of assumptions $A$ such that $P$ is well-typed w.r.t. $A$ and $A$ is data-coherent w.r.t. $P$. Then $\forall e, \forall \tau. A + e : \tau \Rightarrow [P] \vdash_{\text{CRWL}} [e] \rightarrow [\tau]$.

The proofs of all the previous theorems can be found in Appendix A.

## 4. Extensions

### 4.1 Local Definitions

Local definitions (or let-expressions) can allow different degrees of polymorphism to the defined symbols, as explained in [13]. For example local definitions can be completely monomorphic, so all the occurrences of the same defined symbol must have the same type. Therefore an expression like $\text{let F = id in (F true, F z)}$ is ill-typed because $F$ is used with different types: $\text{bool} \rightarrow \text{bool}$ and $\text{nat} \rightarrow \text{nat}$. On the other hand local definitions can be completely polymorphic, so different occurrences of the same defined symbol can have different types provided these types are coherent with the binding. In this case the previous expression is well-typed, as the types of the different appearances of $F$ are instances of the binding type $\alpha \rightarrow \alpha$. Finally, local definitions can have a mixed behavior, considering local definitions as polymorphic if they bind simple variables, or monomorphic if they bind compound patterns. In this case the previous expression is well-typed because the local definition binds a variable—so it behaves polymorphically—whereas it rejects an expression as $\text{let (F, G) = (id, id) in (F true, F z)}$ because $F$ is inside a compound pattern and must be monomorphic.

It is easy to integrate the different kinds of local definitions in our translation. For completely monomorphic local definitions, we only have to remove the binding and generate an equality condition, leaving the rest of the expression untouched. The following example shows the translation considering monomorphic local definitions:

```
\begin{verbatim}
id X = X
f = let (cpair F G) = (cpair id id) in
    \text{cpair (F true, (F false))}
g = let (cpair F G) = (cpair id id) in
    \text{cpair (F true, (F z))}
\end{verbatim}
```

```
t_id = X :\Rightarrow X
t_f = t_cpair \text{ }(F \text{ void} \text{ t}_\text{ tr}) \text{ void} \text{ (F void } t_f) \equiv
    \text{(t_cpair void F void G) =}
    \text{(t_cpair void t_id void t_id)}
t_g = t_cpair \text{ void (F void t_tr) void (F void } t_z) \equiv
    \text{(t_cpair void F void G) =}
    \text{(t_cpair void t_id void t_id)}
\end{verbatim}
```

To extend this translation to polymorphic local definitions we follow the same approach, but repeating as many variants of the binding as occurrences of the defined variables are in the expression. The idea is replacing a polymorphic local definition by several monomorphic local definitions, whose type will be independent because they are used only once. Then we rename every occurrence of a defined variable in the expression to a different variant. For example, the previous function $g$ is translated as:

```
\begin{verbatim}
t_g = t_cpair \text{ void (F void t_tr) void (F prime void t_z)} \equiv
    \text{(t_cpair void F void G) =}
    \text{(t_cpair void t_id void t_id, t_id)}
\end{verbatim}
```

Mixed local declarations are translated using the presented techniques, depending on whether they bind single variables or compound patterns.

### 4.2 Existential Types

Existential types [13 21] is a well-known extension in functional programming which allows a high level of information hiding. This extension allows existential data constructors, having a type $\forall \alpha. \rightarrow (\alpha \rightarrow \text{nat}) \rightarrow (\text{key} \text{X F})$, with the type of the arguments does not occur in the final type), so given a value $\text{key X F}$ we cannot know the exact type of its components $X$ and $F$, only that $X$ has the same type as the argument of $F$. Existential types guarantee that these values whose type is not completely known will never be “inspected”—i.e., matched—and they will never escape the scope. Technically this behavior is obtained by using so-called Skolem constants $\kappa$: when existential constructors appear in the left-hand side of a rule, existential type variables are replaced by different Skolem constants. As Skolem constants are different from any other type this prevents the mentioned “inspections”, both for matching in the left-hand side of a rule or for using it in a context that fix the type in the right-hand side. A final check that Skolem constants do not occur in the returned type of any function—i.e., existential type variables do not escape from the scope—is also needed. For example in the rules $f$ ($\text{key z F} = \text{true}$ and $g$ ($\text{key X F}$) $\equiv X$ the key constructor will have type $\kappa \rightarrow (\kappa \rightarrow \text{nat}) \rightarrow \text{key}$, so the rules are ill-typed as $\kappa$ cannot unify with $\text{nat}$. Similarly, the rule $h$ ($\text{key X F}$) $\equiv X$ is also ill-typed because $X$ has type $\kappa$, which is the returned type.

---

1. We use $\text{ftv}(\cdot)$ to denote the set of free type variables of a type.

---

The proofs of all the previous theorems can be found in Appendix A.
Following these intuitions, we can integrate existential types in our approach. The idea is to add new arguments to existential constructors to represent their existential variables. If existential constructors appear in the left-hand side of a rule, we will pass fresh Skolem types as arguments, otherwise we will use fresh type variables that could be unified to any type. To represent Skolem constants we extend the types data type introduced in Section 3 with a new constructor \( \text{sk} \), of type \( \text{int} \rightarrow \text{types} \). The integer argument will be used to distinguish between different Skolem constants. Consider the following program using the existential constructor \( \text{key} \):

```haskell
data tkey = key A (A \rightarrow \text{nat})
getkey, getkey' :: tkey \rightarrow \text{nat}
getkey (key X F) = F X
getkey' (key z F) = F z

safe = key z s
unsafe (key X F) = X
```

The translation of the data type will produce a rule for evaluating the type of the constructor key, as well as a data constructor tkey in the data type of types:

```haskell
data types = types :-> types | . . . | \text{sk} \text{int} | t\_tkey
t\_key A = A :-> (A :-> t\_nat) :-> t\_tkey
```

Notice that we have added \( A \) as an argument because it is an existential variable, so we will be able to pass Skolem types as arguments when needed. The functions are translated as presented in Section 5, but adding the proper argument to the key constructor:

```haskell
t\_getkey = A <=-
  A == t\_key (sk 1) 00 X 00 F :-> F 00 X,
  not (contains\_sk A)
t\_getkey' = A <=-
  A == t\_key (sk 1) 00 t\_z 00 F :-> F 00 t\_z,
  not (contains\_sk A)
t\_safe = A <=-
  A == t\_key B 00 t\_z 00 t\_s ,
  not (contains\_sk A)
t\_unsafe = A <=-
  A == t\_key (sk 1) 00 X 00 F :-> X,
  not (contains\_sk A)
```

In the rules for getkey, getkey' and unsafe we have passed the Skolem constant \( \text{sk} \) as an argument of the key constructor because it appears in the left-hand side. On the contrary, in the rule of safe we have added a free variable \( B \) because it occurs in the right-hand side, so it can unify with the type \( \text{nat} \). With these definitions, we obtain the expected types: getkey' and unsafe are ill-typed (the first because the Skolem constant \( \text{sk} \) does not unify with \( \text{nat} \), and the second because \( \text{sk} \) 1 appears in the type \( A \), so it escapes from the scope), getkey has type \( \text{tkey} \rightarrow \text{nat} \) and safe has type \( \text{tkey} \):

To avoid these problems, we can use the contains\_sk function to check that the obtained type does not contain Skolem constants. This function needs to traverse types, so the concrete rules depend on the data types defined in the original program. It has to recognize type variables without binding them, i.e., we need a function \( \var \) similar to the predicate \( \var \) of Prolog systems. As happens in the Prolog case, we will define \( \var \) as a pure logic program according to the standard semantics, the function \( \var \) cannot be defined either as a pure functional logic programming according to the standard semantics given by the CRWL framework [8]. Therefore, it is convenient to extend standard functional logic programming with some meta-features similar to the case of Prolog. It is interesting to remark that all the metaprogramming functions that we need can be programmed in the system Toy by using just one built-in impure function \( \text{fails} / 1 \). This function returns true if the argument cannot be evaluated to a head-normal form, and false otherwise. More importantly, it does not produce any variable binding, so we can use it safely to check if an expression is a variable or if it unifies with other expression without performing the unification. Using \( \text{fails} / 1 \) we can define the function \( \var \) — although restricted to arguments of type \( \text{types} \) — because a variable is the only expression that can be unified with two different type values:

```haskell
infix 15 ==:: ==?
X ==:: Y = true <= X==Y
\% equality as true-valued function
X ==?: Y = not (fails (X==Y))
\% returns 'true' if X and Y are unifiable,
\% 'false' otherwise but does not bind them

var :: types -> bool
var X = (X==? t\_boolean) 'and' (X==? t\_nat)
```

### 4.3 Type checking

Up to this point, we have only considered pure inference where the types of all the functions are completely unknown. If (as usual) programmers can provide type declarations for some functions, we need to follow an approach similar to Haskell [20]. First, we translate each type declaration into an auxiliary function that evaluates to the declared type. Then we proceed as before (using the new auxiliary function in all the occurrences of the function) but adding a new condition to check that the inferred type is at least as general as the declared type. The translation can be seen in the following example:

```haskell
f :: list A -> nat
f nil = z
f (cons X Xs) = s (f (cons Xs nil))
f' = t\_list A :-> t\_nat
t\_f = f' <=-
  A == t\_nil :-> t\_z \/ t\_cons 00 X 00 Xs :->
  t\_s 00 (f'@@(t\_cons@@Xs@@t\_nil)),
  f' =:: A
```

We replace all the occurrences of \( f \) by \( f' \) in the right-hand side, and check that the inferred type \( A \) is more general than the declared type \( f' =:: A \). If this condition holds, we return the type \( f' \) declared by the user. The rules of the instance check \( =:: \) depends on the data types in the original program—similar to contains\_sk—but it follows the idea that the type \( t_1 \) is an instance of \( t_2 \) if \( t_2 \) unifies with the result of replacing all the type variables in \( t_1 \) by different type constants—we reuse the Skolem constants \( \text{sk} \) \( N \). Therefore, we use again the \( \var \) function from Section 4.2.

Notice that the previous function uses polymorphic recursion, as \( f \) is used with a more concrete type in the right-hand side.
The type inference of this function will fail without a type declaration, but using it the evaluation returns the expected type (t_list _A) ::= t_nat. However, using our approach it is also possible to perform an iterative type inference as explained in [22] to infer types without type declarations in the presence of polymorphic recursion—see http://gpd.sip.ucm.es/trac/gpd/wiki/TypingFLComputation.

4. Type Errors

The presented translation generates expressions that evaluate to their types or fail if they are ill-typed. Translating functions so that they return a special type error instead of failing in these cases is not easy, as a typing error in sub-expressions must be propagated to the whole expression. For detecting type errors we follow a different philosophy, using a function typeof/1 that given an expression returns its type, or error if it is ill-typed. The code uses again the fails function:

typeof X = if fails (X==X) then error else X

The equality (X==X), whose purpose is forcing the computation of the normal form of X, is important in the definition of typeof. Writing simply typeof X = if fails X then error else X is wrong. For example, the goal typeof (t_twice @@ t_map ) == T would fail instead of returning error as expected. To see this, notice first that fails (t_twice @@ t_map) returns false, because t_twice @@ t_map, although not reducible to normal form, admits reduction to hnf. Therefore, the branch else would be selected in the (wrong) definition of typeof, but then the resulting goal (t_twice @@ t_map) == T fails.

5. Conclusions and Future Work

We have presented (and partially formalized) a wide set of examples developing the following idea: given a functional or functional logic program, build a functional logic program such that evaluation of an expression in it amounts to typing the corresponding expression in the original program. The resulting typing program turns out to be remarkably simple and close to the original one.

Typing programs make use of the integrated nature of functional logic programming: nested functional application helps to preserve the structure of the original program; extra variables (logical variables occurring only on right hand sides) are important to express polymorphism; equational conditions solved by narrowing make it easy to express the constraints involved in type inference or checking.

For the basic cases of Hindley-Milner type inference, pure functional logic programs have been enough to encode typing. However, in the paper we have seen the convenience of considering some meta-functions (for instance, for checking if a term is a variable or if it contains a Skolem type as subterm). These functions lie outside the standard formal semantics of functional logic programming. Not having used them would have implied to come back to a “ground” representation of terms and expressions that would essentially be equivalent to the effort in encoding variables, substitutions and unification that is required in functional programming specifications of type inference. It is interesting to remark that we have been able to program all the needed meta-functions within a standard functional logic language (Toy) by just using one single “impure” (yet operationally easy to understand) feature, namely an already existing primitive function fails/1 that implements non-constructive failure of reduction to head normal form.

Our work has covered only a few cases of type systems. We will try to extend this approach to cover more complex features like intersection types, subtypes or type classes. This might require to consider other kinds of constraints in addition to equational constraints. Functional logic programming inherits from logic programming the ability of incorporating constrains in a natural way. Existing systems include various constraint solving facilities: Herbrand disequality constraints, constraints over real numbers, finite domain constraints, set constraints, and so on, and solid theoretical foundations exist [17]. Thus, we like the idea of recasting the slogan ‘HM(X) is CLP(X) solving’ [27] as ‘HM(X) is CFLP(X) solving’.

We consider that the approach presented in this paper is interesting for several reasons: first, the paper demonstrates vividly the expressive power of functional logic languages, but at the same time it shows also some of their limitations. Besides, the extraction of typing programs from source programs may be applied to the implementation of type inference/checking in different systems, in particular, it could be used as a piece of a bootstrapping compiler for Toy. Due to the clarity of the generated typing programs and the easiness of experimenting (even by hand) with them, the approach has potential educational uses to explain different type inference/checking processes in action.

References

[14] F. López-Fraguas, E. Martín-Martín, and J. Rodríguez-Hortalá. New results on type systems for functional logic programming. In Pro-
cussions of the 18th International Workshop on Functional and (Con-}
• Base cases:
  - \( e \equiv X \) The only possible CRWL-derivation is \( [A] \vdash_{\text{CRWL}} [X] \rightarrow X \) using rule RR, that can be obtained also using \( [P] \).
  - \( e \equiv c \) Trivial by data coherence, which guarantees that the rule for \( e \cdot c \) in \([A]\) and \([P]\) is the same.

• Induction Step:
  - \( e \equiv e_1 \cdot e_2 \) Directly using the Induction Hypothesis.

To prove Theorem 2 we will use the notion of level of an expression, which relies on a topological sort in the dependency graph of the functions. We say a function \( f \) depends on a function \( g \) if \( g \) appears in some rule of \( f \) (in the left or right-hand side). Using those dependencies we can create the strongly connected graph, whose nodes contain sets of functions symbols that depend on each other. Given the strongly connected graph, we can perform a topological sort of the nodes, obtaining a sorted list of nodes such that the function symbols in the \( i^{th} \) node only depend in the nodes from 0 to \( i \). Using this sorted list, we will say that an expression \( e \) has level 1 if the maximum level of its function symbols is 1. Then an expression of level 0 will only contain function symbols in the node 0, or it will contain no function symbols at all.

In the sequel we will denote the union of set of expressions with \( \oplus \) and the usual meaning: \( A \oplus A' \) contains all the assumptions in \( A \) as well as the assumptions in \( A' \) for those symbols not appearing in \( A' \), i.e., new assumptions from \( A' \) replace old assumptions in \( A \).

**Theorem 2** Consider a program \( P \) and a set of assumptions \( A \) such that \( P \) is well-typed w.r.t. \( A \vdash_{\text{wt}} (P) \) and \( A \) is data-coherent w.r.t. \( P \). Then \( \forall e \in \text{Expr} \forall t \in \text{Pat} \cdot [A] \vdash_{\text{CRWL}} [e] \rightarrow t \Rightarrow [P] \vdash_{\text{CRWL}} [e] \rightarrow t \)

**Proof.** By complete induction on \( k = \text{level}(e) \):

• Base case:
  - \( k = 0 \) In this case we proceed by induction on the structure of \( e \). The proof is the same as the one for Lemma 1 but considering the case for function symbols as a base case:

    \( e \equiv f \) For the sake of conciseness we will consider a simple case where level 0 contains two mutually dependent unary functions (\( f \) and \( g \)) with two rules each one, but the proof can be easily extended to any number of functions, rules and larger arities. The rules for these functions are:

    \[
    \begin{align*}
    f & \vdash t_1 \rightarrow e_1 \\
    g & \vdash t_3 \rightarrow e_3
    \end{align*}
    \]

    Since \( \vdash_{A}(P) \) we have that:

    \[
    \begin{align*}
    A & \vdash \lambda t_1. e_1 : \tau_f \rightarrow \tau_f \\
    A & \vdash \lambda t_2. e_2 : \tau_f \rightarrow \tau_f \\
    A & \vdash \lambda t_3. e_3 : \tau_g \rightarrow \tau_g \\
    A & \vdash \lambda t_4. e_4 : \tau_g \rightarrow \tau_g
    \end{align*}
    \]

    According to the definition of well-typed program, each type derivation must obtain a variant of the type \( A(f) \) and \( A(g) \). However, we can easily find a type substitution that joins the types derived for the rules of each function in common types \( \tau_f \rightarrow \tau_f \) and \( \tau_g \rightarrow \tau_g \), (see Theorem A.1-a) in 15, the extended version of 16).

    In each type derivation some assumptions are appended using the rule (A), so we define \( A' \) as the set of assumptions containing all them (note that variables will be different as we assume that the four rules do not have any variable in common). As it is possible to extend the set of assumptions with assumptions for symbols that do not occur in the expression—see Theorem A.1-b) in 15—

    we have the following type derivations:

    \[
    \begin{align*}
    A & \oplus A' \vdash t_1 : \tau_f \\
    A & \oplus A' \vdash t_2 : \tau_f \\
    A & \oplus A' \vdash t_3 : \tau_f \\
    A & \oplus A' \vdash t_4 : \tau_f
    \end{align*}
    \]

    Since polymorphic recursion does not happen, every type sub-derivation for \( f \) or \( g \) in the previous ones will be for the same variant \( \tau_f \rightarrow \tau_f \) and \( \tau_g \rightarrow \tau_g \) respectively. Then we can extend the assumptions \( A'' \equiv A' \oplus \{ A : \tau_f \vdash \tau_f, B : \tau_g \vdash \tau_g \} \) and apply the replacement \( \omega \equiv \{ f \rightarrow A, g \rightarrow B \} \) in the derivations, obtaining the following valid type derivations:

    \[
    \begin{align*}
    A & \oplus A'' \vdash t_1 : \tau_f \\
    A & \oplus A'' \vdash t_2 : \tau_f \\
    A & \oplus A'' \vdash t_3 : \tau_f \\
    A & \oplus A'' \vdash t_4 : \tau_f
    \end{align*}
    \]

    Then by Lemma 1 we have that the following CRWL-derivations are valid:

    \[
    \begin{align*}
    [A] & \vdash_{\text{CRWL}} [t_1] \theta \rightarrow [\tau_f] \\
    [A] & \vdash_{\text{CRWL}} [t_2] \theta \rightarrow [\tau_f] \\
    [A] & \vdash_{\text{CRWL}} [t_3] \theta \rightarrow [\tau_g] \\
    [A] & \vdash_{\text{CRWL}} [t_4] \theta \rightarrow [\tau_g]
    \end{align*}
    \]

    where \( \theta \) substitutes the variables in \( A'' \) by the translations of their types, i.e., if \( A'' \equiv \{ X_n : \tau_n \} \) then \( \theta \equiv \{ X_n \mapsto [\tau_n] \} \).

    The expressions \( t_1 \omega \) and \( t_2 \omega \) are in \( L_{\text{Exp}} \) because \( \text{level}(f) = \text{level}(g) = 0 \) and the only function symbols \( (f \) and \( g) \) have been replaced by \( A \) and \( B \) respectively, so by Lemma 1 we have that \([P] \vdash_{\text{CRWL}} [t_1] \omega \rightarrow [\tau_f], [P] \vdash_{\text{CRWL}} [t_2] \omega \rightarrow [\tau_f] \ldots \)

    The translated program \([P']\) contains the rules:

    \[
    f \cdot g = (A, B) \triangleright \{ [t_1] : \rightarrow [e_1] / [t_2] : \rightarrow [e_2] \} \omega',
    \]

    \[
    \]

    \[
    t \cdot f = A \triangleright \{ A, B \triangleright f, g \}
    \]

    where \( \omega' \equiv \{ t_1, f \rightarrow A, t, g \rightarrow B \} \). Using the previous CRWL-derivations and the OR rule we obtain \([P] \vdash_{\text{CRWL}} f \cdot g \rightarrow [\tau_f], [\tau_g] \) using the substitution \( \theta' \equiv \theta \triangleright [A \rightarrow [\tau_f], B \rightarrow [\tau_g]] \).

    On the other hand, if we have the CRWL-derivation \([A] \vdash_{\text{CRWL}} [f] \rightarrow t \) then \( t \equiv [\tau_f] \rightarrow [\tau_f] \theta \) for some \( \theta \), as \( \tau_f \rightarrow \tau_f \) is a variant of \( A(f) \). Using the OR rule we can easily obtain \([P] \vdash_{\text{CRWL}} [f] \rightarrow [\tau_f] \rightarrow [\tau_f], \) and since CRWL-derivations are closed under substitutions we can also obtain \([\tau_f] \rightarrow [\tau_f] \theta \).

• Induction Step:
  - \( k = j+1 \) The proof is similar to the base case, by induction on the structure of \( c \). The only difference is that we apply the Induction Hypothesis in the case \( e \equiv f \) instead of Lemma 1. Note that we need complete induction because the considered function rules can contain function symbols of any level \( i \leq j \).