Typing as Functional-Logic Evaluation

Francisco J. López Fraguas and Enrique Martin-Martin
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM'13)

Outline

1. Context: functional-logic programming
2. Motivation of the paper
3. Hindley-Milner (HM) typing
4. Extensions to HM
5. Conclusions and future work
Outline

1. Context: functional-logic programming
2. Motivation of the paper
3. Hindley-Milner (HM) typing
4. Extensions to HM
5. Conclusions and future work
**Functional-logic programming**

**Logic programming**
- Non-deterministic search
- Logic variables

**Functional programming**
- Higher-order functions
- Hindley-Milner polymorphism
- Lazy evaluation

**Constraint programming**

**Systems:** Curry (PAKCS, Münster, KICS), **Toy**.
This paper focuses on Toy, which is based on the Constructor Based Rewriting Logic (CRWL) [González-Moreno et al., 1994]

Features of the Toy language:

- Conditional rules
  \[ f \ t_1 \ldots t_n = e \iff e_1 = e'_1, \ldots, e_k = e'_k \]
The *Toy* language

- This paper focuses on *Toy*, which is based on the *Constructor Based Rewriting Logic* (CRWL) [González-Moreno et al., 1994]

- Features of the *Toy* language:
  - Conditional rules
    
    \[ f(t_1 \ldots t_n) = e <= e_1 == e'_1, \ldots, e_k == e'_k \]

  - Constructor terms
This paper focuses on Toy, which is based on the Constructor Based Rewriting Logic (CRWL) [González-Moreno et al., 1994]

Features of the Toy language:

- Conditional rules

\[ f \, t_1 \ldots t_n = e \iff e_1 = e'_1, \ldots, e_k = e'_k \]
This paper focuses on **Toy**, which is based on the *Constructor Based Rewriting Logic* (CRWL) [González-Moreno *et al.*, 1994]

**Features of the Toy language:**

- Conditional rules
  
  \[ f \ t_1 \ldots \ t_n = e \quad \text{<=} \quad e_1 = e'_1, \ldots, e_k = e'_k \]
The *Toy* language

- This paper focuses on *Toy*, which is based on the *Constructor Based Rewriting Logic* (CRWL) [González-Moreno et al., 1994]

- Features of the *Toy* language:
  - Conditional rules
    
    \[ f \; t_1 \ldots t_n = e \iff e_1 = e'_1, \ldots, e_k = e'_k \]
The Toy language

- This paper focuses on Toy, which is based on the Constructor Based Rewriting Logic (CRWL) [González-Moreno et al., 1994]

- Features of the Toy language:
  - Conditional rules
    \[ f \, t_1 \ldots t_n = e \iff e_1 = e'_1, \ldots, e_k = e'_k \]
  - Overlapping rules (non-determinism)
    \[ \text{choice } X \, Y = X \]
    \[ \text{choice } X \, Y = Y \]
The *Toy* language

- This paper focuses on *Toy*, which is based on the *Constructor Based Rewriting Logic* (CRWL) [González-Moreno et al., 1994]

- Features of the *Toy* language:
  - Conditional rules
    \[ f \ t_1 \ldots \ t_n = e \iff e_1 = e'_1, \ldots, e_k = e'_k \]
  - Overlapping rules (non-determinism)
    choice X Y = X
    choice X Y = Y
    Toy> choice true false == L
    \{ L -> true \}
    sol.1, more solutions (y/n/d/a) [y]? y
    \{ L -> false \}
    sol.2, more solutions (y/n/d/a) [y]? y
    no
The *Toy* language

- This paper focuses on *Toy*, which is based on the *Constructor Based Rewriting Logic* (CRWL) [González-Moreno *et al.*, 1994]

- Features of the *Toy* language:
  - Conditional rules
    \[
    f \; t_1 \ldots t_n = e \iff e_1 = e_1', \ldots, e_k = e_k'
    \]
  - Overlapping rules (non-determinism)
    \[
    \text{choice } X \; Y = X \\
    \text{choice } X \; Y = Y
    \]
  - Extra variables (only occur in the right-hand side)
    \[
    \text{sublist } L = \text{Mid } \iff \text{Prev } ++ \; \text{Mid } ++ \; \text{Post } =\; L
    \]
The *Toy* language

- This paper focuses on *Toy*, which is based on the *Constructor Based Rewriting Logic (CRWL)* [González-Moreno et al., 1994]

- Features of the *Toy* language:
  - Conditional rules
    \[
    f \; t_1 \ldots t_n \; = \; e \; <= \; e_1 \; = \; e'_1, \ldots, \; e_k \; = \; e'_k
    \]
  - Overlapping rules (non-determinism)
    \[
    \text{choice } X \; Y \; = \; X \\
    \text{choice } X \; Y \; = \; Y
    \]
  - Extra variables (only occur in the right-hand side)
    \[
    \text{sublist } L \; = \; \text{Mid} \; <= \; \text{Prev} \; ++ \; \text{Mid} \; ++ \; \text{Post} \; = \; L
    \]
The Toy language

- This paper focuses on Toy, which is based on the Constructor Based Rewriting Logic (CRWL) [González-Moreno et al., 1994]

- Features of the Toy language:
  - Conditional rules
    
    \[
    f \ t_1 \ldots \ t_n = e \iff e_1 = e'_1, \ldots, e_k = e'_k
    \]
  - Overlapping rules (non-determinism)
    
    \[
    \text{choice } X \ Y = X \\
    \text{choice } X \ Y = Y
    \]
  - Extra variables (only occur in the right-hand side)
    
    sublist L = Mid <= Prev ++ Mid ++ Post == L

Toy> sublist [1,2] == L
{ L -> [] }
sol.1, more solutions (y/n/d/a) [y]? y
{ L -> [ 1 ] }
sol.2, more solutions (y/n/d/a) [y]? y
{ L -> [ 1, 2 ] }
Outline

1. Context: functional-logic programming
2. Motivation of the paper
3. Hindley-Milner (HM) typing
4. Extensions to HM
5. Conclusions and future work
Motivation

- **Declarative languages** are good candidates to create executable specifications of Hindley-Milner and similar type systems.

- **Functional programming** has been used since long ago. However, it has one main “burden”:
  - Substitutions and unification must be handled explicitly.

- **Logic programming** seems a better candidate because substitutions and unifications are built into the system. However, it usually has some “drawbacks” as the lack of a type system as the FO notation.
Why not using functional-logic programming to specify/implement type systems?

- Since it is a mixture of declarative languages, it takes the best of the previous paradigms for this purpose:
  - Implicit handling of substitutions and unifications,
  - Functional notation (nested applications),
  - Type system in the coding language,
  - Even constraint solving.
This paper

- We propose a **program transformation** to translate a FL program $P$ into another FL typing program $P'$ such that:

  $\text{If } e :: \tau \text{ in } P \text{ then } e' \rightarrow^* \tau \text{ in } P'$

- This translation has some potential uses:
  - It can be used to **implement** different variations of HM type systems,
  - It is remarkably **simple** and **preserves the structure** of the original program. Therefore it can **help students** to understand different type systems,
  - It also shows the **expressive power of FLP**, as well as some of its **limitations**.
Outline

1. Context: functional-logic programming
2. Motivation of the paper
3. Hindley-Milner (HM) typing
4. Extensions to HM
5. Conclusions and future work
In the first approach we will only consider type inference—we don't use or check type declarations—in a HM type system.

We use Toy as the source and object language of the translation.

To simplify, source Toy programs are composed by:

- **Data declarations.** Ex:
  
  data list A = nil | cons A (list A)

- **Type declarations.** Ex:
  
  map :: (A → B) → (list A) → (list B)

- **Function rules.** Ex:
  
  map F (cons X Xs) = cons (F X) (map F Xs)

We omit some other features of the Toy language like operators, type aliases or conditions.
Translation of datatypes

- Since evaluation in the translated program returns types, it must contain a **datatype types** to represent them. Its constructors depend on the original program.

  ```
  data boolean = tr | fl
  data nat = z | s nat
  data list A = nil | cons A (list A)
  ```

- We use the infix constructor ( : ->) to represent functional types.

  ```
  infixr 60 :->
  data types = types :-> types | t_boolean
  | t_nat | t_list types
  ```
Since evaluation in the translated program returns types, it must contain a **datatype types** to represent them. Its constructors depend on the original program.

```
data boolean = tr | fl
data nat = z | s nat
data list A = nil | cons A (list A)
```

```
infixr 60 :->
data types = types :-> types | t_boolean | t_nat | t_list types
```

- We use the infix constructor (: ->) to represent functional types.
Since evaluation in the translated program returns types, it must contain a **datatype types** to represent them. Its constructors depend on the original program.

```hs
data boolean = tr | fl
data nat = z | s nat
data list A = nil | cons A (list A)
```

We use the infix constructor `( : -> )` to represent functional types.

```hs
infixr 60 :->
data types = types :-> types | t_boolean
  | t_nat | t_list types
```
Since evaluation in the translated program returns types, it must contain a **datatype types** to represent them. Its constructors depend on the original program.

```plaintext
data boolean = tr | fl
data nat = z | s nat
data list A = nil | cons A (list A)
```

We use the infix constructor `( :->)` to represent functional types.
Translation of datatypes

- Each data constructor C in the original program generates a nullary function t_C that returns its type.

Source program:
```haskell
data boolean = tr | fl
data nat = z | s nat
data list A = nil | cons A (list A)
```

Translated program:
```haskell
t_tr   = t_boolean
 t_fl  = t_boolean
 t_z    = t_nat
 t_s    = t_nat :-> t_nat
 t_nil  = t_list A
 t_cons = A :-> t_list A :-> t_list A
```
Each data constructor C in the original program generates a nullary function $t_C$ that returns its type.

```haskell
data boolean = tr | fl
data nat = z | s nat
data list A = nil | cons A (list A)
```

```haskell

t_tr = t_boolean

t_fl = t_boolean

t_z = t_nat

t_s = t_nat -> t_nat

t_nil = t_list A

t_cons = A -> t_list A -> t_list A
```
Each data constructor C in the original program generates a nullary function t_C that returns its type.

```haskell
data boolean = tr | fl
data nat = z | s nat
data list A = nil | cons A (list A)
```

- `t_tr = t_boolean`
- `t_fl = t_boolean`
- `t_z = t_nat`
- `t_s = t_nat :-> t_nat`
- `t_nil = t_list A`
- `t_cons = A :-> t_list A :-> t_list A`
Translation of datatypes

- Each data constructor C in the original program generates a nullary function t_C that returns its type.

```haskell
data boolean = tr | fl
data nat = z | s nat
data list A = nil | cons A (list A)
```

```
t_tr   = t_boolean
t_fl   = t_boolean
t_z    = t_nat
```

```
t_s    = t_nat :-> t_nat
```

```
t_nil  = t_list A
```

```
t_cons = A :-> t_list A :-> t_list A
```

Extra variables!!
They can be unified to any type during evaluation (they take the role of universally quantified type variables).
Translation of function rules

- Each function $f$ in the original function generates a nullary function $t_f$ that returns its type.
- This nullary function computes a common type (unification) of all the function rules.

Source program:

```
isEmpty nil = tr
isEmpty (cons X Xs) = fl
```

Translated program:

```
t_isEmpty = (t_nil :-> t_tr)
\\
((t_cons @ X @ Xs) :-> t_fl)
```
Translation of function rules

- Each function $f$ in the original function generates a nullary function $t_f$ that returns its type.
- This nullary function computes a common type (unification) of all the function rules.

```latex
\begin{align*}
\text{isEmpty} \ nil & = \text{tr} \\
\text{isEmpty} \ (\text{cons} \ X \ Xs) & = \text{fl}
\end{align*}
```

```latex
\text{t\_isEmpty} = (\text{t\_nil} :-> \text{t\_tr}) \\
\text{\&\&} \\
((\text{t\_cons} \ @ \ X \ @ Xs) :-> \text{t\_fl})
```
Each function \( f \) in the original function generates a nullary function \( t_f \) that returns its type.

This nullary function computes a common type (unification) of all the function rules.

\[
\begin{align*}
\text{isEmpty } \text{nil} & = \text{tr} \\
\text{isEmpty } \text{(cons } X \text{ Xs)} & = \text{fl}
\end{align*}
\]

\[
t_\text{isEmpty} = (t_\text{nil} :-> t_\text{tr}) \\
\text{\slash}\text{\slash}
((t_\text{cons } @ X @ Xs) :-> t_\text{fl})
\]
Translation of function rules

- Each function \( f \) in the original function generates a nullary function \( t_f \) that returns its type.
- This nullary function computes a common type (unification) of all the function rules.

\[
\begin{align*}
\text{isEmpty} \text{ nil} & = \text{tr} \\
\text{isEmpty} \ (\text{cons} \ X \ Xs) & = \text{fl}
\end{align*}
\]

\[
\text{tIsEmpty} = (\text{tNil} :\rightarrow \text{tTr}) \land ((\text{tCons} @ X @ Xs) :\rightarrow \text{tFl})
\]

\[
\text{infixl 80 @} \\
(\_ :\rightarrow \text{types} \rightarrow \text{types} \rightarrow \text{types}) @ \_'' = \_'' \\
<= \_ = \_''
\]
Each function $f$ in the original function generates a nullary function $t_f$ that returns its type.

This nullary function computes a common type (unification) of all the function rules.

$$\text{isEmpty } \text{nil} = \text{tr}$$
$$\text{isEmpty } \text{(cons } X \text{ } Xs) = \text{fl}$$

$$t_{\text{isEmpty}} = (t_{\text{nil}} :\rightarrow t_{\text{tr}})$$
$$\land$$
$$((t_{\text{cons}} @ X @ Xs) :\rightarrow t_{\text{fl}})$$
Translation of function rules

- Each function $f$ in the original function generates a nullary function $t_f$ that returns its type.
- This nullary function computes a common type (unification) of all the function rules.

\[
\begin{align*}
\text{isEmpty } \text{nil} & = \text{tr} \\
\text{isEmpty } (\text{cons } X \ Xs) & = \text{fl}
\end{align*}
\]

\[
\begin{align*}
t_{\text{isEmpty}} & = (t_{\text{nil}} :\rightarrow t_{\text{tr}}) \\
& \quad \wedge \\
& \quad ((t_{\text{cons}} @ X @ Xs) :\rightarrow t_{\text{fl}})
\end{align*}
\]

\[
\text{infixr} \ 50 \ \wedge \\
(\wedge) :: \text{types} \rightarrow \text{types} \rightarrow \text{types} \\
T \ \wedge \ T' = T \leq T \equiv T'
\]
Translation of function rules

- Each function $f$ in the original function generates a nullary function $t_f$ that returns its type.
- This nullary function computes a common type (unification) of all the function rules.

```haskell
isEmpty nil         = tr
isEmpty (cons X Xs) = fl

t_isEmpty = (t_nil :-> t_tr) /
            ((t_cons @ X @ Xs) :-> t_fl)

Toy> t_isEmpty == T
    { T -> (t_list _A) :-> t_boolean } 
    sol.1, more solutions (y/n/d/a) [y]? y
    no
```

```haskell
t_isEmpty = (t_nil :-> t_tr) 
    /
    ((t_cons @ X @ Xs) :-> t_fl)
```
Recursive functions:

Replace each recursive call with the same extra variable, which represents the type of the function.

\[
\begin{align*}
\text{map } F \text{ nil} & = \text{ nil} \\
\text{map } F \text{ (cons } X \text{ Xs)} & = \text{ cons } (F \text{ } X) \text{ (map } F \text{ } Xs)
\end{align*}
\]

\[
\text{t_map} = A \iff A == \\
(F_1 : \rightarrow t_{\text{nil}} : \rightarrow t_{\text{nil}} \\
\land \\
F_2 : \rightarrow t_{\text{cons}} @ X_2 @ Xs2 : \rightarrow t_{\text{cons}} @ (F_2 @ X_2) @ (A @ F_2 @ Xs2))
\]
Recursive functions:

Replace each recursive call with the same extra variable, which represents the type of the function.

map F nil = nil
map F (cons X Xs) = cons (F X) (map F Xs)

\[
t_{\text{map}} = A \iff A == \\
(F1 :-> t_{\text{nil}} :-> t_{\text{nil}} \\
\text{\textbackslash} \\
F2 :-> t_{\text{cons}} @ X2 @ Xs2 :-> t_{\text{cons}} @ (F2 @ X2) @ (A @ F2 @ Xs2))\
\]
Recursive functions:

Replace each recursive call with the same extra variable, which represents the type of the function.

\[
\begin{align*}
\text{map } F \text{ nil} &= \text{nil} \\
\text{map } F \text{ (cons } X \text{ Xs) } &= \text{cons } (F \ X) \ (\text{map } F \ Xs)
\end{align*}
\]

we must use variants for each rule.

\[
\begin{align*}
t\_\text{map} &= A \iff A \iff \\
&(F1 \mapsto t\_\text{nil} :\mapsto t\_\text{nil} \\
&\backslash \\
&F2 \mapsto t\_\text{cons } @ \ X2 \ @ \ Xs2 :\mapsto t\_\text{cons } @ (F2 \ @ \ X2) \ @ (A \ @ \ F2 \ @ \ Xs2))
\end{align*}
\]
Translation of function rules (II)

- **Recursive functions:**
  - Replace each recursive call with the same extra variable, which represents the type of the function.

```haskell
Toy> t_map == T
{ T -> (_A :-> _B) :-> (t_list _A) :-> (t_list _B) }
sol.1, more solutions (y/n/d/a) [y]? y
  no

t_map = A <=< A ==
(F1 :-> t_nil :-> t_nil /
F2 :-> t_cons @ X2 @ Xs2 :-> t_cons @ (F2 @ X2) @ (A @ F2 @ Xs2))
```
Translation of function rules (III)

- Mutually dependent functions:
  - An auxiliary nullary function infers the types of the functions at the same time: it returns a tuple of types
  - The typing function simple projects the tuple

\[
\begin{align*}
\text{odd } z &= \text{fl} \\
\text{odd } (s \ X) &= \text{even } X \\
\text{even } z &= \text{tr} \\
\text{even } (s \ X) &= \text{odd } X
\end{align*}
\]

\[
\text{odd_even} = (A,B) \iff
\begin{align*}
A &= (t_z \rightarrow t_{\text{fl}} \ \land \ t_s @ X \rightarrow B @ X), \\
B &= (t_z \rightarrow t_{\text{tr}} \ \land \ t_s @ X_1 \rightarrow A @ X_1)
\end{align*}
\]

\[
\begin{align*}
t_{\text{odd}} &= A \iff (A,B) = \text{odd_even} \\
t_{\text{even}} &= B \iff (A,B) = \text{odd_even}
\end{align*}
\]
Mutually dependent functions:

- An auxiliary nullary function infers the types of the functions at the same time: it returns a tuple of types.
- The typing function simple projects the tuple.

\[
\begin{align*}
\text{odd } z &= \text{fl} \\
\text{odd } (s \ X) &= \text{even } X \\
\text{even } z &= \text{tr} \\
\text{even } (s \ X) &= \text{odd } X
\end{align*}
\]

\[
\text{odd_even} = (A,B) \iff
\begin{align*}
A &= (t_z :\rightarrow t_{\text{fl}} \land t_s @ X :\rightarrow B @ X), \\
B &= (t_z :\rightarrow t_{\text{tr}} \land t_s @ X_1 :\rightarrow A @ X_1)
\end{align*}
\]

\[
\begin{align*}
t_{\text{odd}} &= A \iff (A,B) = \text{odd_even} \\
t_{\text{even}} &= B \iff (A,B) = \text{odd_even}
\end{align*}
\]
Mutually dependent functions:

An auxiliary nullary function infers the types of the functions at the same time: it returns a tuple of types.

The typing function simple projects the tuple.

\[
\begin{align*}
\text{odd } z &= \text{fl} \\
\text{odd } (s \; X) &= \text{even } X
\end{align*}
\]

Toy> \( t_{\text{odd}} = \text{TODD}, \; t_{\text{even}} = \text{TEVEN} \)
\[
\{ \text{TODD} \rightarrow t_{\text{nat}} :\rightarrow t_{\text{boolean}}, \quad \text{TEVEN} \rightarrow t_{\text{nat}} :\rightarrow t_{\text{boolean}} \} 
\]
sol.1, more solutions (y/n/d/a) [y]? y
no

\[
\begin{align*}
\text{odd\_even} &= (A,B) \iff \\
A &= (t_z :\rightarrow t_{\text{fl}} \land t_s @ X :\rightarrow B @ X), \\
B &= (t_z :\rightarrow t_{\text{tr}} \land t_s @ X1 :\rightarrow A @ X1)
\end{align*}
\]

\[
\begin{align*}
n_{\text{odd}} &= A \iff (A,B) = \text{odd\_even} \\
n_{\text{even}} &= B \iff (A,B) = \text{odd\_even}
\end{align*}
\]
Adequacy of the translation

- For the source program, we consider a type system very similar to HM proposed in [López-Fraguas et al., 2012]
  - $\mathcal{A} \vdash e : \tau$ means that, under the set of assumptions $\mathcal{A}$, the expression $e$ has type $\tau$
- As semantics we use *Higher-Order* extension of *CRWL* (HO-CRWL) [González-Moreno et al., 1997]
  - $\mathcal{P} \vdash_{CRWL} e \rightarrow t$ denotes that, under the program $\mathcal{P}$, $t$ is a value of $e$.
- Then if a program $\mathcal{P}$ is well-typed w.r.t. a set of assumptions $\mathcal{A}$ we have:

  **Adequacy**

  If $\mathcal{A} \vdash e : \tau$ then $\mathcal{P}' \vdash_{CRWL} e' \rightarrow \tau'$
1. Context: functional-logic programming
2. Motivation of the paper
3. Hindley-Milner (HM) typing
4. Extensions to HM
5. Conclusions and future work
Extensions to HM

- The presented translation can be adapted to cover some extension of the HM type system.
- I'll only show typechecking, but the paper covers also existential typed, local definitions and type errors.
- The adapted translations are still simple, but they need metaprogramming functions (impure).
Typechecking

- Up to now, we have only considered *type inference* without using the type declarations for functions.
- It is possible to adapt our approach to use function type declarations (**typechecking**). We need:
  - **Translate** the function *type declarations* into new auxiliary functions.
  - A final check to ensure that the *inferred types* for functions are at least as general as the declared ones.
Typechecking (I)

- A type declaration for the function \( f \) is translated into the function \( f' \) (in the obvious way).

\[
f :: \text{list A} \rightarrow \text{nat}
\]

\[
f' = \text{t_list A} :\rightarrow \text{t_nat}
\]
Typechecking (II)

- Function rules are translated as before, but:
  - Each call to a function \( f \) with a type declaration is translated into \( f' \) (instead of \( t_f \)): we use the declared type instead of the inferred one.
  - We add a final check to ensure that the inferred type is as general as the declared one

\[
\begin{align*}
f & : \text{list } A \rightarrow \text{nat} \\
f \text{ nil} & = z \\
f \text{ (cons } X \text{ Xs)} & = s (f \text{ Xs})
\end{align*}
\]

\[
\begin{align*}
f' & = t_{\text{list } A} : \rightarrow t_{\text{nat}} \\
t_f & = f' <=
A & = t_{\text{nil}} : \rightarrow t_{\text{z}} \setminus (t_{\text{cons} @X @Xs}) : \rightarrow t_{s} @ (f' @Xs), \\
f' & = :< A
\end{align*}
\]
Typechecking (II)

- Function rules are translated as before, but:
  - Each call to a function \( f \) with a type declaration is translated into \( f' \) (instead of \( t\_f \)): we use the declared type instead of the inferred one.
  - We add a final check to ensure that the inferred type is as general as the declared one.

\[
\begin{align*}
  f & :\text{ list } A \rightarrow \text{ nat} \\
  f\ \text{nil} & = z \\
  f\ (\text{cons } X\ \text{Xs}) & = s\ (f\ \text{Xs})
\end{align*}
\]

\[
f' = t\_\text{list } A :\rightarrow t\_\text{nat}\\
t_f = f' \iff A == t\_\text{nil} :\rightarrow t\_z \land (t\_\text{cons}\@X@\text{Xs}) :\rightarrow t\_s @ (f'@\text{Xs}),
\]

Since \( f \) has a type declaration, we use \( f' \) instead of \( t\_f \).
Typechecking (II)

- Function rules are translated as before, but:
  - Each call to a function \( f \) with a type declaration is translated into \( f' \) (instead of \( t_f \)): we use the declared type instead of the inferred one.
  - We add a final check to ensure that the inferred type is as general as the declared one.

\[
\begin{align*}
f &:: \text{list } A \rightarrow \text{nat} \\
f \text{ nil} &\ = \ z \\
f (\text{cons } X \ Xs) &\ = \ s (f \ Xs)
\end{align*}
\]

\[
\begin{align*}
f' &\ = \ t\_\text{list } A :\rightarrow \ t\_\text{nat} \\
t_f &\ = \ f' \ <== \\
A &\ = \ t\_\text{nil} :\rightarrow \ t\_z \ \& \ (t\_\text{cons}@X@Xs) :\rightarrow \ t\_s @ (f'@Xs), \\
f' &\ \ = \ <\ A
\end{align*}
\]

Metaprogramming
- We check that the inferred type (\( A \)) is as general as the declared one (\( f' \)).
Typechecking (II)

- Function rules are translated as before, but:
  - Each call to a function $f$ with a type declaration is translated into $f'$ (instead of $t_f$): we use the declared type instead of the inferred one.
  - We add a final check to ensure that the inferred type is as general as the declared one:

$$
\begin{align*}
  f &:: \text{list } A \rightarrow \text{nat} \\
  f \text{ nil} & = z \\
  f \text{ (cons } X \text{ Xs)} & = s (f \text{ Xs})
\end{align*}
$$

Finally, we ensure that the type returned is the declared one:

$$
\begin{align*}
  f' & = t_{\text{list } A} :\rightarrow t_{\text{nat}} \\
  t_f & = f' \iff \\
  A & = t_{\text{nil}} :\rightarrow t_{z} \setminus (t_{\text{cons } X@Xs} :\rightarrow t_{s} @ (f'@Xs)), \\
  f' & = :\leq A
\end{align*}
$$
Outline

1. Context: functional-logic programming
2. Motivation of the paper
3. Hindley-Milner (HM) typing
4. Extensions to HM
5. Conclusions and future work
Conclusions

- We have developed a **program transformation** from a FL program $P$ to another FL program $P'$ such that:

  $e :: \tau$ in $P$ then $e' \rightarrow^* \tau'$ in $P'$

- It is remarkably **simple**, and **preserve the structure** of applications of the original program. Then it has **potential educational uses** to explain different type inference/checking processes to students

- It can be applied to the **FL-implementation** of the **type stage** in a FL compiler
Conclusions

- The simplicity of the translation shows the expressive power of FLP (in this case, the use of extra variables, unification and functional application).

- The translation can be extended to cover other aspects like typechecking, existential types, local definitions or type errors. Unfortunately, many of them need metaprogramming functions (impure), so their correctness cannot be proved with the standard semantics of FLP semantics.
Future work

- Extend the approach to **cover more complex features** like:
  - Subtyping
  - Generalized Algebraic Data Types (GADTs)
  - Type classes
- Study if the **constraint solving** integrated into FLP can be valuable (pursuing a slogan like to 'HM(X) is CFLP(X) solving')
Thank you!