Advances in Type Systems for Functional Logic Programming
(Work in progress)

Francisco J. López-Fraguas    Enrique Martin-Martín
Juan Rodríguez-Hortalá

Dpto. Sistemas Informáticos y Computación
Universidad Complutense de Madrid

IX Jornadas sobre Programación y Lenguajes
PROLE 2009

San Sebastián, 8-11 de Septiembre, 2009
Functional Logic Programming

- **Declarative programming**: The programmer establishes what are the properties of the solution instead of how to find it.

<table>
<thead>
<tr>
<th>Functional Programming</th>
<th>Logic Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>HO functions</td>
<td>Logic variables</td>
</tr>
<tr>
<td>Type system</td>
<td>Search for solutions</td>
</tr>
</tbody>
</table>

Ex: *Haskell, ML, Clean*  
Ex: *Prolog*

Functional Logic Languages: *TOY, Curry, Escher, Mercury...*
**FLP Example**

**last**: returns the last element of a list

\[
\begin{align*}
\text{append } [] & \text{ Ys } = \text{ Ys} \\
\text{append } (X:Xs) & \text{ Ys } = X : (\text{append } Xs \text{ Ys}) \\
\text{last } L & = E \iff (\text{append } Xs \text{ [E]}) = L
\end{align*}
\]

**Examples:**

- last [1,2,3] → 3
- last [true] → true
- last [] → no
This work

We propose a type system tackling two orthogonal aspects:

1) A naive application of a Damas & Milner type system in a language with HO patterns may produce problems from the point of view of types [GHR01].

2) There are various grades of polymorphism for bound variables in let expressions.
Motivation and informal description
HO patterns

Higher order pattern
Pattern with partial application of constructors or functions.
Ex: id, map id, and true
- They appear naturally in the semantics.
- They are useful as intensional representations of functions.

```plaintext
type point = (real, real)
type region = point -> bool

empty :: region
rect :: point -> point -> region
empty P = false
rect P1 P2 P = (...)

union :: region -> region -> region
union empty R P = R P
union R1 R2 P = (R1 P) or (R2 P)
```
Type problems with HO patterns

Example ([GHR01] & Curry mailing list)

\[
\begin{align*}
\text{snd} & : A \rightarrow B \rightarrow B \\
\text{snd} \ X \ Y & \rightarrow Y \\
\text{cast} & : A \rightarrow B \\
\text{cast} \ X & \rightarrow \text{unpack} \ (\text{snd} \ X) \\
\text{unpack} & : (A \rightarrow A) \rightarrow B \\
\text{unpack} \ (\text{snd} \ X) & \rightarrow X \\
\text{and} & : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} \\
\text{and} \ \text{true} \ X & \rightarrow X \\
\text{and} \ \text{false} \ X & \rightarrow \text{false}
\end{align*}
\]
The expression \( \text{and} \ (\text{cast} \ 0) \ \text{true} : \text{bool} \) is well-typed, since \((\text{cast} \ 0) : \text{bool}\).

If we apply the rules of \text{cast} and \text{unpack} we obtain:
\[
\text{and} \ (\text{cast} \ 0) \ \text{true} \xrightarrow{\text{cast}} \text{and} \ (\text{unpack} \ (\text{snd} \ 0)) \ \text{true} \xrightarrow{\text{unpack}} \text{and} \ 0 \ \text{true}
\]

\(\text{and} \ 0 \ \text{true} \) is ill-typed

\[\Downarrow\]

Well-typed programs can go wrong!
Where is the problem?

In function \textit{unpack}. Knowing the type of its pattern \((\text{snd } X)\) does not provide any information about the type of its subpattern \(X\).

Damas & Milner type systems treat this “opacity” as polymorphism, and this is the reason why they infer a type \((A \rightarrow A) \rightarrow B\) for \textit{unpack}.
Type problems with HO patterns (IV)

How to solve this situation?

a) Solution of [GHR01]: opaque patterns are forbidden. 

\( t \) is opaque iff it contains \( f \ t_1 \ldots t_n \) s.t. \( f : \overline{\tau_n} \rightarrow \tau \) and 
\( FTV(\overline{\tau_n}) \not\subseteq FTV(\tau) \).

*Examples:* With \( \text{snd} : A \rightarrow B \rightarrow B \)
How to solve this situation?

a) Solution of [GHR01]: opaque patterns are forbidden.  
\( t \) is opaque iff it contains \( f \ t_1 \ldots t_n \) s.t. \( f : \overline{\tau_n} \rightarrow \tau \) and \( FTV(\overline{\tau_n}) \not\subseteq FTV(\tau) \).

**Examples:** With \( \text{snd} \) : \( A \rightarrow B \rightarrow B \)
- \( \text{snd true} \) \( \times \)
How to solve this situation?

a) Solution of [GHR01]: opaque patterns are forbidden. 
   \( t \) is opaque iff it contains \( f \ t_1 \ldots t_n \) s.t. \( f : \overline{\tau_n} \rightarrow \tau \) and 
   \( FTV(\overline{\tau_n}) \notin FTV(\tau) \).

Examples: With \( \text{snd} \):

- \( \text{snd \ true} \) ✗
- \( \text{snd \ zero} \) ✗
How to solve this situation?

a) Solution of [GHR01]: opaque patterns are forbidden. 

\( t \) is opaque iff it contains \( f \ t_1 \ldots t_n \) s.t. \( f : \overline{\tau_n} \rightarrow \tau \) and \( FTV(\overline{\tau_n}) \not\subseteq FTV(\tau) \).

**Examples:** With \( \text{snd} \): \[
\begin{array}{ccc}
A & \rightarrow & B \rightarrow B \\
\text{snd true} & \times \\
\text{snd zero} & \times \\
\text{snd X} & \times 
\end{array}
\]
Type problems with HO patterns (V)

b) **Our solution**: Making a distinction between *transparent* and *opaque* variables. A variable of a pattern is transparent if its type is *univocally* fixed by the type of the pattern, and opaque otherwise.

We reject only opaque variables when they appear in the rest of the expression, i.e., when they are *critical*.

- In the example, $X$ is opaque in $(\text{snd } X)$ and appears in the right-hand side of the rule for *unpack*, so the program will be rejected.
- However, patterns like $\text{snd true}$ or $\text{snd zero}$ have not any opaque variable, so they would be accepted.
Another generalization for free

We allow **opaque data constructors**, i.e., constructors with type
\[ \bar{\tau}_n \rightarrow \tau \text{ s.t. } FTV(\bar{\tau}_n) \not\subseteq FTV(\tau). \]

Example: \( \text{cont} : \alpha \rightarrow \text{container} \)

These data constructors are not allowed in existing FP or FLP systems.

In our framework we treat them the same way as HO patterns.
Opaque data constructors are useful

```haskell
data container = cont A  % cont : A → container

code :: container -> string
code (cont false)     = "000"
code (cont true)      = "001"
code (cont zero)      = "010"
code (cont (succ X))  = "011" ++ code (cont X)
```

This program is not legal in [GHR01] or FLP systems.
Our second contribution: polymorphism of let expressions

There exist different grades of polymorphism for variables in let expressions.

Different implementations have different grades of polymorphism. They do not usually document their choice.

We explicitly formalize 3 kinds of let expressions

\[
\text{let } t = e_1 \text{ in } e_2
\]

in our type system.
Monomorphic lets

\[ \text{let}_m t = e_1 \text{ in } e_2 \]

All the variables in \( t \) have a monomorphic type.

Systems: Clean 2.0, \( \mathcal{T}OY \) 2.3.1\(^1\), PAKCS 1.9.1, KICS 0.81893.

Examples:

- \( \text{let } F = id \text{ in } (F 0, F 4) \) ✓
- \( \text{let } F = id \text{ in } (F \text{ true}, F 0) \) ✗
- \( \text{let } [F, G] = [id, id] \text{ in } (F \text{ true}, F 0, G 0, G \text{ false}) \) ✗

\(^1\text{monomorphic where expressions} \)
Polymorphic lets

\[
\text{let}_p t = e_1 \text{ in } e_2
\]

All the variables in the pattern have a polymorphic type.


Examples:

- let \( F = id \) in \((F \ 0, F \ 4)\)
- let \( F = id \) in \((F \ true, F \ 0)\)
- let \([F, G] = [id, id]\) in \((F \ true, F \ 0, G \ 0, G \ false)\)
“Mixed” lets

\[
\text{let}_{pm} \ t = e_1 \ in \ e_2
\]

The polymorphism depends on the form of the pattern:

- a) If the pattern is a **single variable**, it is **polymorphic**.
- b) If the pattern is **compound**, the variables are **monomorphic**.

Systems: GHC 6.8.2, Standard ML of New Jersey 110.67, Curry Münster 0.9.11.

Examples:

- \( \text{let } F = \text{id} \ in \ (F \ 0, F \ 4) \) ✓
- \( \text{let } F = \text{id} \ in \ (F \ \text{true}, F \ 0) \) ✓
- \( \text{let } [F, G] = [\text{id}, \text{id}] \ in \ (F \ \text{true}, F \ 0, G \ 0, G \ \text{false}) \) ✗
Formal aspects
We have divided the type system into two parts:

a) A basic typing relation $\vdash$ to give a type to an expression.

b) An extended typing relation $\vdash^*$ that uses the previous and also checks the absence of critical variables.
Basic typing relation $\vdash$

$\mathcal{A} \vdash e : \tau$

$\mathcal{A}$ - set of assumptions over symbols: $\{s_i : \sigma_i\}$

$e$ - expression

$\tau$ - simple type
Basic typing relation $\vdash$: rules (I)

**[ID]**

\[
\frac{}{\mathcal{A} \vdash s : \tau} \quad \text{if} \quad s \in DC \cup FS \cup DV \\
\land (s : \sigma) \in \mathcal{A} \land \sigma \succ \tau
\]

**[APP]**

\[
\frac{\mathcal{A} \vdash e_1 : \tau_1 \rightarrow \tau}{\mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1e_2 : \tau}
\]

**[Λ]**

\[
\frac{\mathcal{A} \oplus \{X_i : \tau_i\} \vdash t : \tau_t}{\mathcal{A} \vdash \lambda t.e : \tau_t \rightarrow \tau} \quad \text{if} \quad \{X_i\} = var(t)
\]

\[
\frac{\mathcal{A} \oplus \{X_i : \tau_i\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \rightarrow \tau} \quad \text{if} \quad \{X_i\} = var(t)
\]
Basic typing relation $\vdash$: rules (II)

**[LET$_m$]**

\[
\begin{align*}
\mathcal{A} \oplus \{X_i : \tau_i\} & \vdash t : \tau_t \\
\mathcal{A} & \vdash e_1 : \tau_t \\
\mathcal{A} \oplus \{X_i : \tau_i\} & \vdash e_2 : \tau_2 \\
\mathcal{A} & \vdash \text{let}_m \ t = e_1 \ \text{in} \ e_2 : \tau_2
\end{align*}
\]

if $\{X_i\} = \text{var}(t)$

**[LET$_p$]**

\[
\begin{align*}
\mathcal{A} \oplus \{X_i : \tau_i\} & \vdash t : \tau_t \\
\mathcal{A} & \vdash e_1 : \tau_t \\
\mathcal{A} \oplus \{X_i : \text{Gen}(\tau_i, \mathcal{A})\} & \vdash e_2 : \tau_2 \\
\mathcal{A} & \vdash \text{let}_p \ t = e_1 \ \text{in} \ e_2 : \tau_2
\end{align*}
\]

if $\{X_i\} = \text{var}(t)$

$\text{Gen}(\tau, \mathcal{A})$ is the closure or generalization of $\tau$ wrt. $\mathcal{A}$

Formally: $\text{Gen}(\tau, \mathcal{A}) = \forall \alpha_i. \tau$ where $\{\alpha_i\} = FTV(\tau) \setminus FTV(\mathcal{A})$. 
Basic typing relation ⊢: rules (III)

\[ \text{[LET}^X_{pm}] \]
\[
\begin{align*}
\mathcal{A} & \vdash e_1 : \tau_1 \\
\mathcal{A} \oplus \{X : \text{Gen}(\tau_1, \mathcal{A})\} & \vdash e_2 : \tau_2 \\
\mathcal{A} & \vdash \text{let}_pm X = e_1 \text{ in } e_2 : \tau_2
\end{align*}
\]

\[ \text{[LET}^h_{pm}] \]
\[
\begin{align*}
\mathcal{A} \oplus \{X_i : \tau_i\} & \vdash h \ t_1 \ldots t_n : \tau_t \\
\mathcal{A} & \vdash e_1 : \tau_t \\
\mathcal{A} \oplus \{X_i : \tau_i\} & \vdash e_2 : \tau_2 \\
\mathcal{A} & \vdash \text{let}_pm h \ t_1 \ldots t_n = e_1 \text{ in } e_2 : \tau_2
\end{align*}
\]
Opaque and critical variables

Opaque variable

“A variable of a pattern is **opaque** if its type is not **univocally** fixed by the type of the pattern.”

Critical variable

“A variable is **critical** if it is opaque in a pattern of a lambda or let expression and appears in the rest of the expression.”

Examples:

- $X$ is opaque $\text{snd } X$
- $X$ is not opaque in $\text{snd } [X, \text{true}]$
- $X$ is critical in $\lambda(\text{snd } X).X$
- $X$ is not critical in $\lambda(\text{snd } X).\text{true}$
Extended typing relation \( \vdash \)

\[ \mathcal{A} \vdash \bullet \; e : \tau \]

\( \mathcal{A} \) - set of assumptions over symbols

\( e \) - expression

\( \tau \) - simple type

Forbids only expressions with critical variables.

\[ [P] \quad \frac{\mathcal{A} \vdash e : \tau}{\mathcal{A} \vdash \bullet \; e : \tau} \quad \text{if } \text{critVar}_\mathcal{A}(e) = \emptyset \]
Subject reduction

- **Well-typed program** \((wt_A(P))\). \(P\) is well-typed wrt. \(A\) iff for every rule \(f \ t_1 \ldots t_n \rightarrow e \in P\):
  a) \(A \vdash \lambda t_1 \ldots t_n . e : \tau\)
  b) \(\tau\) is a variant of \(A(f)\)

- *Let*-rewriting semantics \([LRS08]\)

---

**Theorem (Subject reduction)**

*If* \(A \vdash e : \tau\) *and* \(wt_A(P)\) *and* \(P \vdash e \rightarrow^l e'\) *then* \(A \vdash e' : \tau\).
Type inference for expressions

\[ \mathcal{A} \vdash e : \tau | \pi \quad \mathcal{A} \vdash \cdot e : \tau | \pi \]

\( \mathcal{A} \) - set of assumptions
\( e \) - expression
\( \tau \) - most general type for \( e \)
\( \pi \) - minimum type substitution which is needed to apply to \( \mathcal{A} \) in order to derive a type for \( e \)

Intuition:

a) wherever type derivation guesses types, type inference introduces fresh type variables.

b) wherever type derivation forces equality of types, type inference computes a mgu.

c) infer a variant for symbols.
Properties of the type inference for expressions

Theorem (Soundness of type inference)

\[ \mathcal{A} \models e : \tau|\pi \quad \Rightarrow \quad \mathcal{A}\pi \vdash e : \tau \]
\[ \mathcal{A} \models \bullet e : \tau|\pi \quad \Rightarrow \quad \mathcal{A}\pi \vdash \bullet e : \tau \]

Theorem (Completeness of \( \models \) wrt. \( \vdash \))

If \( \mathcal{A}\pi' \vdash e : \tau' \) then
\[ \exists \tau, \pi, \pi''. \mathcal{A} \models e : \tau|\pi \land \mathcal{A}\pi\pi'' = \mathcal{A}\pi' \land \tau\pi'' = \tau'. \]

Theorem (Maximality of \( \models \bullet \))

a) \[ \bullet \Pi^e_{\mathcal{A}} \equiv \{ \pi \in TSubst \mid \exists \tau \in SType. \mathcal{A}\pi \vdash \bullet e : \tau \} \text{ has a maximum element} \]
\[ \iff \exists \tau_g \in SType, \pi_g \in TSubst. \mathcal{A} \models \bullet e : \tau_g|\pi_g. \]

b) If \( \mathcal{A}\pi' \vdash \bullet e : \tau' \) and \( \mathcal{A} \models \bullet e : \tau|\pi \) then exists a type substitution \( \pi'' \) such that \( \mathcal{A}\pi' = \mathcal{A}\pi\pi'' \) and \( \tau' = \tau\pi''. \)
Type inference for Programs

- **Block inference.** Procedure $B$ to infer types for programs without explicit type declarations.

$$B(A, \{rule_1, \ldots, rule_m\}) = \pi$$

**Theorem (Soundness of $B$)**

If $B(A, \mathcal{P}) = \pi$ then $wt_{A\pi}(\mathcal{P})$.

**Theorem (Maximality of $B$)**

If $wt_{A\pi'}(\mathcal{P})$ and $B(A, \mathcal{P}) = \pi$ then $\exists \pi''$ such that $A\pi' = A\pi\pi''$.

- **Stratified inference.** Divides the program in mutually dependent blocks, infer types and generalizes.
Conclusions

When compared to [GHR01] we make the following contributions:

- Consideration of local pattern bindings and $\lambda$-abstractions.
- Formalization of different possibilities for polymorphism in local bindings.
- Subject reduction is proved wrt. a small-step operational semantics ($let$-rewriting), closer to real computations.
- Algorithms for inferring types for programs without explicit type declarations.
Future Work

- Integrate the type inference for programs into the TOY compiler.
- Generalize the subject reduction property to narrowing, using let-narrowing reductions.
- Study the possible relation between opacity and existential types or GADTs.
- Extend the type system to support type classes and extra variables in the right-hand sides.
End
J.C. González-Moreno, T. Hortalá-González, and Rodríguez-Artalejo, M.
Polymorphic types in functional logic programming.
Special issue of selected papers contributed to the International Symposium on Functional and Logic Programming (FLOPS’99).

Rewriting and call-time choice: the HO case.
The problem with narrowing

\[ \text{id } X = X \]
\[ \text{and true } X = X \]
\[ \text{and false } X = \text{false} \]

- \( F \text{ zero } == \text{ zero } \leadsto [F/id] \text{ true } \)
- \( F \text{ zero } == \text{ zero } \leadsto [F/\text{and true}] \text{ true } \)
  and true zero is \textbf{ill-typed}