Introducción al tema de los sistemas de tipos en programación funcional lógica. Se presentan avances en los sistemas de tipos para programación funcional lógica.

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

Dpto. Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Spain

18th Int'l Workshop on Functional and (Constraint) Logic Programming
WFLP 2009

Brasília, Brazil - 28 June, 2009
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 [González-Moreno et al., 2001].

2) There are various grades of polymorphism for bound variables in let expressions.
Motivation and informal description
Higher order pattern: pattern with partial application of constructors or functions
   Examples: id, map id, snd X, and true

Example (Curry mailing list)

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>snd</td>
<td>A → B → B</td>
<td>co</td>
<td>(A → A) → B</td>
</tr>
<tr>
<td>snd X Y</td>
<td>Y</td>
<td>co (snd X)</td>
<td>X</td>
</tr>
<tr>
<td>cast</td>
<td>A → B</td>
<td>and</td>
<td>bool → bool → bool</td>
</tr>
<tr>
<td>cast X</td>
<td>co (snd X)</td>
<td>and true X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and false X</td>
<td>false</td>
</tr>
</tbody>
</table>
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 \( \text{co} \) we obtain:
\[
\text{and}(\text{cast} \ 0) \ \text{true} \xrightarrow{\text{cast}} \ \text{and}(\text{co} (\text{snd} \ 0)) \ \text{true} \xrightarrow{\text{co}} \ \text{and} \ 0 \ \text{true}
\]

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

\[\Downarrow\]

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

In function \textit{co}. 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{co}. 
How to solve this situation?

a) Solution of [González-Moreno et al., 2001]: opaque patterns are forbidden. \( t \) is opaque iff it contains \( f t_1 \ldots t_n \) s.t. 
\[
 f : \overline{\tau_n} \rightarrow \tau \quad \text{and} \quad FTV(\overline{\tau_n}) \notin FTV(\tau).
\]

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 \((\text{snd } X)\) and appears in the right-hand side of the rule for \( co \), so the program will be rejected.
Another generalization for free

We allow **opaque data constructors**, i.e., constructors with type $\overline{\tau_n} \rightarrow \tau$ s.t. $FTV(\overline{\tau_n}) \not\subseteq FTV(\tau)$.

Example: $cont : \alpha \rightarrow container$

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

In our framework we treat them the same way as HO patterns.
Example

```haskell
data container = cont A  % cont : A \to container

code :: container \to string

code (cont false) = "000"

code (cont true) = "001"

code (cont zero) = "010"

code (cont (succ X)) = "011" ++ code (cont X)
```

All these patterns are rejected in [González-Moreno et al., 2001]
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, TOY 2.3.1*, PAKCS 1.9.1, KICS 0.81893.

Examples:

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

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

All the variables in the pattern have a polymorphic type.


Examples:

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

- \(\text{let} \ F = \text{id} \ \text{in} \ (F \ 0, F \ 4)\) ✓
- \(\text{let} \ F = \text{id} \ \text{in} \ (F \ \text{true}, F \ 0)\) ✓
- \(\text{let} \ [F, G] = [\text{id}, \text{id}] \ \text{in} \ (F \ \text{true}, F \ 0, G \ 0, G \ \text{false})\) ✗
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 $\vdash^\bullet$ that uses the previous and also checks the absence of critical variables.
Basic typing relation $\vdash$.

$A \vdash e : \tau$

$A$ - set of assumptions over symbols: $\{s_i : \sigma_i\}$
$e$ - expression
$\tau$ - simple type
Basic typing relation $\vdash$: rules (I)

\begin{align*}
\textbf{[ID]} & \quad \frac{} {\mathcal{A} \vdash s : \tau} \quad \text{if } s \in \text{DC} \cup \text{FS} \cup \text{DV} \land (s : \sigma) \in \mathcal{A} \land \sigma \succ \tau \\
\textbf{[APP]} & \quad \frac{\mathcal{A} \vdash e_1 : \tau_1 \rightarrow \tau \quad \mathcal{A} \vdash e_2 : \tau_1} {\mathcal{A} \vdash e_1 e_2 : \tau} \\
\textbf{[Λ]} & \quad \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 } \{X_i\} = \text{var}(t)
\end{align*}
Basic typing relation $\vdash$: rules (II)

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

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

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

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 $\{ \alpha_i \} = \text{FTV}(\tau) \setminus \text{FTV}(\mathcal{A})$. 

Basic typing relation \( \vdash : \) rules (III)

\[
\begin{align*}
\text{[LET}_{pm}^X] & \quad \mathcal{A} \vdash e_1 : \tau_1 \\
& \quad \mathcal{A} \oplus \left\{ X : \text{Gen}(\tau_1, \mathcal{A}) \right\} \vdash e_2 : \tau_2 \\
& \quad \mathcal{A} \vdash \text{let}_{pm} X = e_1 \text{ in } e_2 : \tau_2 \\
\text{[LET}_{pm}^h] & \quad \mathcal{A} \oplus \left\{ X_i : \tau_i \right\} \vdash h \, t_1 \ldots t_n : \tau_t \\
& \quad \mathcal{A} \vdash e_1 : \tau_t \\
& \quad \mathcal{A} \oplus \left\{ X_i : \tau_i \right\} \vdash e_2 : \tau_2 \\
& \quad \mathcal{A} \vdash \text{let}_{pm} h \, t_1 \ldots t_n = e_1 \text{ in } e_2 : \tau_2 \\
& \quad \text{if } \left\{ X_i \right\} = \text{var}(t_1 \ldots t_n) \\
& \quad \wedge h \in DC \cup FS
\end{align*}
\]
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 $\vdash^\bullet$

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

- $\mathcal{A}$ - set of assumptions over symbols
- $e$ - expression
- $\tau$ - simple type

Forbids only expressions with \textbf{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 [López-Fraguas et al., 2008]

**Theorem (Subject reduction)**

If \(A \vdash \bullet e : \tau\) and \(wt_A(P)\) and \(P \vdash e \rightarrow^l e'\) then \(A \vdash \bullet e' : \tau\).
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 \textbf{fresh type variables}.

b) wherever type derivation forces equality of types, type inference computes a \textbf{mgu}.

c) infer a \textbf{variant} for symbols.
Properties of the type inference for expressions

Theorem (Soundness of type inference)
\[ \mathcal{A} \vdash e : \tau | \pi \implies \mathcal{A} \pi \vdash e : \tau \quad \mathcal{A} \vdash \ast e : \tau | \pi \implies \mathcal{A} \pi \vdash \ast e : \tau \]

Theorem (Completeness of $\vdash$ wrt. $\vdash$)
If $\mathcal{A} \pi' \vdash e : \tau'$ then
\[ \exists \tau, \pi, \pi''. \mathcal{A} \vdash e : \tau | \pi \land \mathcal{A} \pi \pi'' = \mathcal{A} \pi' \land \tau \pi'' = \tau'. \]

Theorem (Maximality of $\vdash \ast$)
\begin{enumerate}
\item $\ast \Pi_{\mathcal{A}}^e$ has a maximum element
\[ \iff \exists \tau_g \in SType, \pi_g \in TSubst. \mathcal{A} \vdash \ast e : \tau_g | \pi_g. \]
\item If $\mathcal{A} \pi' \vdash \ast e : \tau'$ and $\mathcal{A} \vdash \ast e : \tau | \pi$ then exists a type substitution $\pi''$ such that $\mathcal{A} \pi' = \mathcal{A} \pi \pi''$ and $\tau' = \tau \pi''$.
\end{enumerate}
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 \( B(A, P) = \pi \) then \( wt_{A\pi}(P) \).

**Theorem (Maximality of \( B \))**

If \( wt_{A\pi'}(P) \) and \( B(A, P) = \pi \) 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 [González-Moreno et al., 2001] 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 (\textit{let}-rewriting), closer to real computations.
- Algorithms for inferring types for programs without explicit type declarations.
Future Work

- Implement the type inference for programs to be used in the \texttt{TOY} compiler.
- Generalize the subject reduction property to narrowing, using \textit{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
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.
In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS’08)*, volume 4989 of LNCS, pages 147–162. Springer.