Cleaner semantics for Objective Label

Jacques Garrigue

October 26, 1999

Credits

This proposal contains ideas from Damien Doligez and Pierre Weis.

Labels and optionals

Labels and optional arguments had two problems in Objective Label.

- They were not fully coherent with the original call-by-value semantics of the language. In some (subtle) cases, a side-effect might get delayed more than in an untyped semantics.
- For optional arguments, no untyped semantics existed.

This new proposal corrects these two flaws.

Syntax

We keep Objective Label's syntax, except for default values in optional arguments.

$$
\begin{array}{rcl}
\textit{typexpr} & ::= & \ldots \\
 & | & \textit{typexpr} \rightarrow \textit{typexpr} \\
 & | & [?]\mathbf{label}{:}\textit{typexpr} \rightarrow \textit{typexpr} \\
\textit{expr} & ::= & \ldots \\
 & | & \textit{expr}\ \textit{labeled-expr}^{+} \\
 & | & \texttt{fun}\ \{\textit{labeled-simple-pattern}\}^{+}\ [\texttt{when}\ \textit{expr}] \rightarrow \textit{expr} \\
 & | & \texttt{function}\ \textit{labeled-pattern}\ [\texttt{when}\ \textit{expr}] \rightarrow \textit{expr} \\
 & & \quad \{\texttt{|}\ \textit{labeled-pattern}\ [\texttt{when}\ \textit{expr}] \rightarrow \textit{expr}\}^{*} \\
\textit{labeled-expr} & ::= & [?]\textit{expr} \\
 & | & [?]\mathbf{label}{:}\textit{expr} \\
\textit{labeled-pattern} & ::= & \textit{pattern} \\
 & | & \mathbf{label}{:}\textit{pattern} \\
 & | & ?[(\textit{expr})]\mathbf{label}{:}\textit{pattern}
\end{array}
$$

Dynamic semantics

$\emptyset$ is a notation for the empty label.

$$
\begin{array}{ll}
(\texttt{fun}\ \mathbf{l_i}{:}x \rightarrow e)\ \mathbf{l_1}{:}e_1 \ldots \mathbf{l_n}{:}e_n & \\
\qquad \rightarrow (e[e_i/x]\ \mathbf{l_1}{:}e_1 \ldots \mathbf{l_{i-1}}{:}e_{i-1}\ \mathbf{l_{i+1}}{:}e_{i+1} \ldots \mathbf{l_n}{:}e_n & \\
(\texttt{fun}\ ?\mathbf{l_i}{:}x \rightarrow e)\ \mathbf{l_1}{:}e_1 \ldots \mathbf{l_n}{:}e_n & \\
\qquad \rightarrow e[\texttt{Some}(e_i)/x]\ \mathbf{l_1}{:}e_1 \ldots \mathbf{l_{i-1}}{:}e_{i-1}\ \mathbf{l_{i+1}}{:}e_{i+1} \ldots \mathbf{l_n}{:}e_n & \\
(\texttt{fun}\ ?\mathbf{l}{:}x \rightarrow e)\ \mathbf{l_1}{:}e_1 \ldots \mathbf{l_n}{:}e_n & \text{when } \mathbf{l_i} = \emptyset \text{ and } l \notin \{\mathbf{l_1} \ldots \mathbf{l_n}\} \\
\qquad \rightarrow e[\texttt{None}/x]\ \mathbf{l_1}{:}e_1 \ldots \mathbf{l_n}{:}e_n & \\
((\texttt{fun}\ \mathbf{l}{:}x \rightarrow e)\ \mathbf{l_1}{:}e_1 \ldots \mathbf{l_m}{:}e_m)\ \mathbf{l_{m+1}}{:}e_{m+1} \ldots \mathbf{l_n}{:}e_n & \text{when } \mathbf{l} \notin \{\mathbf{l_1} \ldots \mathbf{l_m}\} \\
\qquad \rightarrow (\texttt{fun}\ \mathbf{l}{:}x \rightarrow e)\ \mathbf{l_1}{:}e_1 \ldots \mathbf{l_n}{:}e_n & \\
((\texttt{fun}\ ?\mathbf{l}{:}x \rightarrow e)\ \mathbf{l_1}{:}e_1 \ldots \mathbf{l_m}{:}e_m)\ \mathbf{l_{m+1}}{:}e_{m+1} \ldots \mathbf{l_n}{:}e_n & \text{when } \{\mathbf{l}, \emptyset\} \not\subset \{\mathbf{l_1} \ldots \mathbf{l_m}\} \\
\qquad \rightarrow (\texttt{fun}\ ?\mathbf{l}{:}x \rightarrow e)\ \mathbf{l_1}{:}e_1 \ldots \mathbf{l_n}{:}e_n &
\end{array}
$$

Typing

Semantics are kept throughout compilation by disallowing label commutation for function types. However, the original comfort of out-of-order application is recovered by allowing argument reordering in application, when the function's type is *well known* (*c.f.* polymorphic methods).

Variants

Variant typing, as it is presented in the user's manual, is not principal: in some cases typability of an expression may depend on the order in which the typing algorithm proceeds.

```
# let f1 (x : [< a b(int)]) = ()
  let f2 (x : [< a]) = ()
  let f3 (x : [< a b(bool)]) = ();;
val f1 : [< a b(int)] -> unit = <fun>
val f2 : [< a] -> unit = <fun>
val f3 : [< a b(bool)] -> unit = <fun>
# fun x -> f1 x; f2 x; f3 x;;
- : [< a] -> unit = <fun>
# fun x -> f1 x; f3 x;;
Characters 18-19:
This expression has type [< a b(int)] but is here used with type
  [< a b(bool)]
```

Here the constraint introduced by `f2` hides the constructor `b`, and avoids a clash between `int` and `bool`.

An easy way to solve this would be to restrict hiding absent labels to generic types. This way the second case would still fail, since `x` has a monorphic type. This solution would be correct and principal.

However, one can easily see that this solution is counter-intuitive. For the user, `b` is already an impossible constructor, and having a clash on it is hard to understand.

Another solution is to go the opposite way. To accept more programs. This is the way we explore here, with an unchanged syntax.

Typing

The idea is to delay unification on constructor until they are explicitely known to be present. We keep the (T, U, L) representation of variant types, but T is no longer a map from constructors to types, but from constructors to sets of types.

When we unify two variant types, the first step is just to take the union of both typing environments, dropping unnecessary types.

$$(T_1, U_1, L_1) \wedge (T_2, U_2, L_2) = ((T_1|_{U_1 \cap U_2}) \cup (T_2|_{U_1 \cap U_2}), U_1 \cap U_2, L_1 \cup L_2)$$

Here the union of two typing environments is the pointwise union of their sets of types for each constructor.

This first step never fails.

In a second step, structural constraints are enforced on the resulting type (T, U, L). First, L should be included in U. Then, for all constructors appearing in L, the set of types associated with each constructor is collapsed by unification. This can be expressed by rewriting rules, where e is a multi-equation and ϕ a set of multiequations

$$\text{if } L \not\subset U \text{ then } (T, U, L) = e \wedge \phi \longrightarrow \bot$$

$$\begin{array}{l}\text{if } l \in L \text{ and } T(l) = \{\tau_1, \ldots, \tau_n\} \text{ then} \\ \quad (T, U, L) = e \wedge \phi \longrightarrow (T\{l \mapsto \tau_1\}, U, L) = e \wedge \tau_1 = \ldots = \tau_n \wedge \phi\end{array}$$

Optionally one can add rules that remove a constructor l from U if the equation obtained from $T(l)$ has no solution. Such rules would be sound and complete.

Syntax of types

Thanks to the good properties of these constraints, the surface syntax of types would only have to be slightly extended.

$$\begin{array}{rcl} \textit{tag-type} & ::= & \textit{ident} \\ & | & \textit{ident}\ \texttt{(}\textit{typexpr-list}\texttt{)} \\ \textit{typexpr-list} & ::= & \textit{typexpr} \\ & | & \textit{typexpr}\ \texttt{\&}\ \textit{typexpr-list} \end{array}$$

Notice that a 0-ary constructor and an 1-ary construtor are contradictory, and would result in the absence of this constructor.

Discussion

Such a change has the major advantage of both recovering principality and avoiding unintuitive error messages. Constraints created in such a way are very light: they always appear inside a variant type, and if the variant type does not appear in the final type scheme, then the constraint can be discarded safely.

On the other hand, there are two drawbacks.

- Some errors will be delayed longer than now, until a construtor is actually included in L. It is not clear how damageable it is.
- While type inference is simple and costless for this extension, simplification of constraints —marking constructors with unsolvable constraints as absent, and eliminating redundant types in constraints— is a bit more expensive. Also, allowing such constrained types inside signatures would mean having to solve a matching problem, which is exponential in the number of connected constraints inside a type scheme.

Reasonably efficient algorithms exist to solve these problems, so the difficulty is more in the increased complexity of the type-checker than in run-time cost.

Other features

Objective Label contains two other features: polymorphic methods and type-driven access of records. Both of them use the same method of enforcing principality of typing through tracing user provided type information. With this tracing, their implementation is very easy, but without it they loose principality.

While these features provide some comfort in writing user programs, they are not strictly necessary for the various libraries coming with O'Labl (LablTk, LablGL and LablGTK).