Unions of Abstract Polymorphic Variants

Romain Bardou
Internship in Nagoya University (Japan), with Jacques Garrigue
March – August 2006
Synopsis

- Abstract polymorphic variants
- Unions
- Motivations for unions
- Checking compatibilities
Abstract polymorphic variants
Variants

Variants are sum types with labels.

```plaintext
type expr =
    Int of int |
    Plus of expr * expr

let example1 = Plus(Int 27, Int 42)
```

They have to be declared first.
Polymorphic Variants

Polymorphic Variants types are inferred.

```
let example2 = 'Plus('Int 27, 'Int 42)
```

```
val example2 : 
  [> 'Plus of [> 'Int of int ] * [> 'Int of int ] ]
```

This means `example2` has at least the label `Plus`.
Subsumption

One can instantiate polymorphic variants.

```ocaml
let example3 = 'A

val example3 : ['A | 'B | 'C of int ] = 'A

(example3 : ['A | 'B | 'C of int ])

- : ['A | 'B | 'C of int ] = 'A
```
Abstract Polymorphic Variants

Use the `private` keyword.

```ocaml
module type Expr = sig
    type expr = private [> 'Plus of expr * expr ]
    val un: expr
    val eval: expr -> int
end

module IntExpr: Expr = struct
    type expr = [ 'Int of int | 'Plus of expr * expr ]
    let un = 'Int 1
    let rec eval = ... 
end
```
Abstraction

IntExpr.eval ('Plus(IntExpr.un, IntExpr.un))

- : int = 2

IntExpr.eval ('Int 10)

  IntExpr.eval ('Int 10)

This expression has type [> ‘Int of int ] but is here used with type
  IntExpr.expr
Unions
Concrete polymorphic variants can be used in other definitions.

\[
\text{type intexpr} = \text{[ ‘Int of int ]}
\]
\[
\text{type boolexpr} = \text{[ ‘Bool of bool ]}
\]
\[
\text{type expr} = \text{[ intexpr | boolexpr ]}
\]

\[
\text{type expr} = \text{[ ‘Bool of bool | ‘Int of int ]}
\]

The expansion is done immediately.
Unions of abstract types

The following code raises an error at compilation.

```ocaml
module A: sig
    type intexpr = private [> ]
    type boolexpr = private [> ]
end = (...

    type expr = [ A.intexpr | A.boolexpr ]
```

Indeed, there is no way to check whether this union is safe, as not all labels are known.
The actual implementations of \texttt{intexpr} and \texttt{boolexpr} could be incompatible.

\begin{verbatim}
module A = struct
  type intexpr = ['Item of int]
  type boolexpr = ['Item of bool]
end

type expr = [A.intexpr | A.boolexpr]

expr would associate both int and bool to 'Item.
\end{verbatim}
Summary

- Polymorphic variants
  - No declaration, no collision on labels
  - Enhanced modularity
  - Locating errors is harder

- Private types
  - Semi-abstraction
  - Great for functors
  - No union
Motivation for unions

Building a language in a modular fashion. We start by defining small pieces of the language.

```ocaml
module type Expr = sig
  type t = private [> ]
end
module Int = struct
  type t = [ 'Int of int ]
end
module Bool = struct
  type t = [ 'Bool of bool ]
end
```
We then define a functor which combines languages.

```ocaml
module Mix(A: Expr)(B: Expr) = struct
    type t = [ A.t | B.t ]
end
```

Note how $t$ makes an union of two abstract polymorphic variants.

Now, as both `Int` and `Bool` have the signature `Expr`, we can combine them in a single language.

```ocaml
module IntBool = Mix(Int)(Bool)
```
Functions using these abstract types could also be defined.

```ocaml
module type Expr = sig
  type t = private [> ]
  val show: t -> string
end
module Int = struct
  type t = [ 'Int of int ]
  let show = function 'Int i -> string_of_int i
end
module Bool = struct
  type t = [ 'Bool of bool ]
  let show = function 'Bool b -> string_of_bool b
end
```
The operator #, which already exists for concrete polymorphic variants, could then be used.

```ocaml
module Mix(A: Expr)(B: Expr): Expr = struct
  type t = [ A.t | B.t ]
  let show = function
    #A.t as x -> A.show x
    | #B.t as x -> B.show x
end

module IntBool = Mix(Int)(Bool)

IntBool.show ('Int 1), ^IntBool.show ('Bool true)

- : string = "1, true"
```
The idea is to add compatibility information.

```plaintext
module A: sig
    type intexpr = private [> ]
    type boolexpr = private [> ] ~[ intexpr ]
end = (…)

type expr = [ A.intexpr | A.boolexpr ]
```

A.boolexpr is said to be compatible with A.intexpr, allowing expr to be defined.
Other kinds of compatibilities:

```ocaml
type t1 = private [> ] ~[ 'Shared of int ]
type u1a = [ t1 | 'Shared of int ]
type u1b = [ t1 | 'Shared of bool ] (* error *)

type t2 = private [> ] ~[ ~'Shared ]
type u2a = [ t2 | 'Shared of int ]
type u2b = [ t2 | 'Shared of bool ]

type t3 = private [> ] ~[ ~t ]
type u3 = [ t3 | t ]
```
Summary

- Unions between private types
  - Compatibility information on private type definitions
  - Extension of #t in pattern-matching

- Problems
  - Checking compatibilities
  - Preserving type inference
Checking compatibilities
Validity test

For each definition such as:

\[
\text{type } t = \text{private } [> P_1 \mid \ldots \mid P_n ] \sim [ C_1 \mid \ldots \mid C_n ]
\]

1. Check if \( P_i \odot P_j \) for all \( i, j \)

2. Check if \( P_i \odot C_j \) for all \( i, j \)

3. Add \( t \) and its definition to an environment \( \Theta \)
Compatibility relation (LL)

Checking the compatibility of two labels is easy:

\[
\frac{l \neq l' \text{ or } \tau = \tau'}{\Theta \vdash l \text{ of } \tau \odot l' \text{ of } \tau'}^{LL}
\]
Compatibility relation (TT)

Two types are compatible if one uses the other:

\[ \Theta \vdash t \supseteq t' \]

\[ \Theta \vdash t \odot t' \]

or if there is an explicit compatibility:

\[ \Theta \vdash \text{?}t \supseteq t' \]

\[ \Theta \vdash t \odot t' \]
Compatibility relation (LT)

Similarly to Type / Type compatibility:

\[
\begin{align*}
\Theta &\vdash l \text{ of } \tau \circ t \quad \text{LT1} \\
\Theta &\vdash l \text{ of } \tau \odot t \\
\Theta &\vdash l \text{ of } \tau \ominus t \\
\Theta &\vdash l \text{ of } \tau \otimes t \\
\Theta &\vdash l \text{ of } \tau \ominus t
\end{align*}
\]

but absence information can be used too:

\[
\Theta \vdash l \text{ of } \tau \ominus t
\]
Consider the following example:

\[
\text{type } \tau = \text{private } [> \ ] [ 'A \text{ of int } | 'A \text{ of bool }]
\]

The only way \( \tau \) can be compatible with two different types for \( 'A \) is if \( \tau \) doesn’t use \( 'A \).

This leads to:

\[
\Theta \vdash ?l \text{ of } \tau_1 \odot \tau \quad \Theta \vdash ?l \text{ of } \tau_2 \odot \tau \quad \tau_1 \neq \tau_2 \quad \text{LT4}
\]

\[
\Theta \vdash l \text{ of } \tau \odot \tau
\]
Inheritance relation

$A \subseteq B$ when $B$ inherits $A$.

This is read from the environment.

Another option is to expand type names before checking compatibilities.
Inheritance: base case

\[ R \in \Theta(t) \]
\[ \Theta \vdash R \leftarrow t \]

In1
Inheritance of presence information

Example:

```
type t = private [\> 'A of int ]
type u = private [\> t ]
```

u inherits ‘A of int.

\[
t \in \Theta(u) \quad \Theta \vdash A \sqsubseteq t \\
\Theta \vdash A \sqsubseteq u
\]
Inheritance of compatibilities

Example:

\[
\begin{align*}
\text{type } t &= \text{private } [> \text{ `A of int } ] \\
\text{type } u &= \text{private } [> ]~[ t ]
\end{align*}
\]

\(u\) is compatible with \(\text{`A of int}\).

\[
\begin{array}{c}
\Rightarrow t \in \Theta(u) \\
\Theta \vdash A \otimes t \\
\Theta \vdash ?A \otimes u \\
\end{array}
\]

\(\text{In3} \)
Conclusion

We proposed an extension to private polymorphic variants to handle unions, using compatibility annotations.

We modeled types and proved our compatibility relation is sound and complete \textit{w.r.t.} our model.

A prototype implementation is available as a branch of OCaml.