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.

```
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.

```
let example3 = ‘A

val example3: [> ‘A ] = ‘A

(example3 : [ ‘A | ‘B | ‘C of int ])

- : [ ‘A | ‘B | ‘C of int ] = ‘A
```

Abstract Polymorphic Variants

Use the `private` keyword.

```
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

Unions

Concrete polymorphic variants can be used in other definitions.

```
type intexpr = [ ‘Int of int ]
type boolexpr = [ ‘Bool of bool ]
type expr = [ intexpr | boolexpr ]

type expr = [ ‘Bool of bool | ‘Int of int ]
```

The expansion is done immediatly.

Unions of abstract types

The following code raises an error at compilation.

```
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 `intexpr` and `boolexpr` could be incompatible.

```
module A = struct
  type intexpr = [ ‘Item of int ]
  type boolexpr = [ ‘Item of bool ]
end

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

type expr = [ ‘Item of int | ‘Item of bool ]
```

`expr` would associate both `int` and `bool` to `‘Item`.

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 be defining small pieces of the language.

```
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.

```
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.

```
module IntBool = Mix(Int)(Bool)
```

Functions using these abstract types could also be defined.

```
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.

```
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"
```

Compatibility information

The idea is to add compatibility information.

```
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:

```
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:

```
type t = private [> P1 | ... | Pn ]~[ C1 | ... | Cn ]
```

1. Check if `Pi` $\odot$ `Pj` for all i, j

2. Check if `Pi` $\odot$ `Cj` for all i, j

3. Add t and its definition to an environment Θ

Compatibility relation (LL)

Checking the compatibility of two labels is easy:

$$\frac{l \neq l' \text{ or } \tau = \tau'}{\Theta \vdash l \texttt{ of } \tau \odot l' \texttt{ of } \tau'}\mathsf{LL}$$

Compatibility relation (TT)

Two types are compatible if one uses the other:

$$\frac{\Theta \vdash t \circledin t'}{\Theta \vdash t \odot t'}\text{TT1}$$

or if there is an explicit compatibility:

$$\frac{\Theta \vdash ?t \circledin t'}{\Theta \vdash t \odot t'}\text{TT1}$$

Compatibility relation (LT)

Similarly to Type / Type compatibility:

$$\frac{\Theta \vdash l \texttt{ of } \tau \Subset t}{\Theta \vdash l \texttt{ of } \tau \odot t}\text{LT1} \qquad \frac{\Theta \vdash ?l \texttt{ of } \tau \Subset t}{\Theta \vdash l \texttt{ of } \tau \odot t}\text{LT2}$$

but absence information can be used too:

$$\frac{\neg l \in \Theta(t)}{\Theta \vdash l \texttt{ of } \tau \odot t}\text{LT3}$$

Consider the following example:

```
type t = private [> ]~[ `A of int | `A of bool ]
```

The only way `t` can be compatible with two different types for $`A$ is if `t` doesn't use $`A$.

This leads to:

$$\frac{\Theta \vdash ?l \texttt{ of } \tau_1 \circledcirc t \quad \Theta \vdash ?l \texttt{ of } \tau_2 \circledcirc t \quad \tau_1 \neq \tau_2}{\Theta \vdash l \texttt{ of } \tau \odot t}\textsf{LT4}$$

Inheritance relation

$A \circledcirc B$ when B *inherits* A.

This is read from the environment.

Another option is to expand type names before checking compatibilities.

Inheritance: base case

$$\frac{R \in \Theta(t)}{\Theta \vdash R \circledin t}\text{In1}$$

Inheritance of presence information

Example:

```
type t = private [> ‘A of int ]
type u = private [> t ]
```

`u` inherits `‘A of int`.

$$\frac{t \in \Theta(u) \qquad \Theta \vdash A \circledin t}{\Theta \vdash A \circledin u}\textsf{In2}$$

Inheritance of compatibilities

Example:

```
type t = private [> ‘A of int ]
type u = private [> ]~[ t ]
```

`u` is compatible with `‘A of int`.

$$\frac{?t \in \Theta(u) \qquad \Theta \vdash A \circledin t}{\Theta \vdash ?A \circledin u}\text{In3}$$

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 *w.r.t.* our model.

A prototype implementation is available as a branch of OCaml.