Value Restriction について (参考資料)

おおいわ

May 1, 2001

前回までに型多相と reference などの変更可能フィールドについて説明しましたが、実はこの 2 つは相性が悪いことがわかっています。次のような (架空の) 例を考えてみます。

```
# let r1 = ref (fun x -> x);;
val r1 : ('a -> 'a) ref = {contents=<fun>}
# let f () = (!r1 true, !r1 5);;
val f : unit -> bool * int = <fun>
# r1 := (fun x -> x + 1);;
- : unit = ()
# f ();;
```

この一連のセッションの最後の関数適用は、明らかに `(fun x -> x + 1)` を `true` に適用しているので型エラーです。この原因はどこにあるのでしょう。

もちろんこれは $(\alpha \to \alpha)$ のフィールドに `(int → int)` の値を代入しているのが根本的原因なのですが、単純に型システムをこの代入を禁止するように変更するだけでは解決できません。

```
# let twice f x = f (f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
# let set_r1_twice f x = (r1 := f; !r1 (!r1 x));;
val set_r1 : ('a -> 'a) -> 'a -> 'a = <fun>

# twice (fun x -> x + 1) 5;;
- : int = 7
# set_r1_twice (fun x -> x) 5;;
- : int = 5
# set_r1_twice (fun x -> x + 1) 5;;
- : int = 7
# !r1 true;;
```

ここで、r1 の型が $(\alpha \to \alpha)$ `ref` だとすると、`twice` も `set_r1_twice` の型は一致します。しかし、`set_r1_twice` の方だけこっそり中で r1 に値を保存していますので、同じ問題を引き起こします。

このような問題になる例をいろいろ考えていくと、結局のところ、型整合をとりつつ参照に多相型を与えるのは困難だという結論に達します。そのため、OCaml ではとりあえず、参照型のオブジェクトを多相型をもちそうな方法で定義すると、多相型ではなくて「未決定の単相型」をシステムが与えます。

```
# let r1 = ref (fun x -> x);;
val r1 : ('_a -> '_a) ref = {contents=<fun>}
# let f1 () = !r1 true;;
val f1 : unit -> bool = <fun>
# let f2 () = !r1 5;;
This expression has type int but is here used with type bool
# r1 := (fun x -> x + 1);;
This expression has type (bool -> bool) ref but is here used with type
(int -> int) ref
# !r1;;
- : bool -> bool = <fun>
```

この例の 1 行目で、r1 の型には未決定の単相型 '_a が現れています。2 行目の引数と型チェックをすると、この '_a が bool であることが確定するので、以降この r1 を別の型で使おうとするとエラーになります。結果として、4 行目のような既にある f1 と相容れない値は型チェッカで拒絶され、型安全性が守られます。

では、このような単相型を生成するのは変更可能フィールドが明示的に出てきた場合だけでいいのでしょうか。実はそうもいかないことがわかっています。unit $\to \alpha \to \alpha$ 型の値を () に適用した場合を考えます。最も単純な (fun () -> (fun x -> x)) の場合、返り値の型は $\alpha \to \alpha$ で問題ありません。しかし、次のような関数を考えると、単相型である必要があります。

```
let make_delay () =
  let r = ref [] in
  let delay x =
    let old =
      match !r with
        []   -> x (* no previous element *)
      | y::_ -> y (* previous element *)
    in r := [x]; old
  in delay
```

この関数は、初回には与えられた引数を、2 回目以降はその前の回に与えられた値を記憶しておいて返すような関数を作ります。返される delay は単相的にしか使えませんが、make_delay 自体は多相的です。

結局、構文的に副作用のない即値であると確実にわかるもの以外に多相型を与えるのは危険だということになります。このことを value restriction と呼んでいます。

多相的	定数式, fun 式, それらの値の tuple, それらの値の変更不可能なデータ構造
単相的	変更可能データ構造, let 式、関数適用の結果, etc...

例えば、式 (fun () -> (fun x -> x)) () の結果は、式全体が関数適用の形をしているので、単相型 '_a → '_a となります。

なお、この結果として、高階関数の部分適用が予期せぬ単相型を生成することがあります。この場合、引数宣言を追加して全体を関数 (fun, λ 式) の形にすることで回避できます。このような変換は一般に η 展開と呼ばれる操作です。

```
# let mappair' = map (fun x -> (x, x));;
val mappair' : '_a list -> ('_a * '_a) list = <fun>
# let mappair l = map (fun x -> (x, x)) l;;
val mappair : 'a list -> ('a * 'a) list = <fun>
```

また、未決定の単相型の解決は、今回の本編で説明する 1 つの structure の中でのみ行なわれます。これは、一般にモジュールは分割コンパイルの単位なので、2 つ以上のモジュールで未決定の単相型を用いた際に問題が発生するためです。