mf werkexemplaar

# Hiding of auxiliary functions in Miranda-abstract types

Maarten Fokkinga, 18 october 1985.

The abstract type facility of Miranda is based on transparency rather than hiding, that is: all the identifiers involved in the "definition" of the abstract type are visible throughout the script. However, sometimes one needs (for efficiency and/or comprehensability, indeed even for theoretical necessity [Bergstra 1980]) auxiliary functions in the definition; and of course these should be hidden for the user of the abstract type. We show that it is ~~very~~ easy to achieve this in Miranda.

* * *

Throughout this note we use the Miranda notation, see [Turner 1985]. Consider as an example of an abstract type the well-known stack. Here is its definition.

→ stack *

```
abstype stack *
with   empty :: stack *;          isempty :: stack * → bool;
       push :: * → stack *;       pop :: stack * → stack *;
       top :: stack * → *;
|| comment: ~~this~~ here ends the abstype - construct.
```

```
stack *  ==  [*]              || list-of-*
empty  =  []
isempty x  =  (x = [])
push a x  =  a:x
pop (a:x)  =  x
top (a:x)  =  a
```

As you see, all identifiers mentioned in the signature of stack * (that is the part following with) are accessible throughout the remainder of the script (= program). The problem thus reads: what to do with auxiliary functions, say functions f and g, which would be used in the definition of empty, isempty, push, pop and top but which should not be visible to the user? Where should we define them? What would their types look like?

The solution is simple: the scope rules provide the means to restrict visibility. ~~Thus~~ So f and g should be defined in a where-clause which is attached to the definitions of empty, ... , top jointly. As in Miranda where-clauses may only be added to single expressions (of arbitrary complexity), this forces us to invent a single expresion defining all functions empty , ... , top. This is easy :

```
(empty, isempty, push, pop, top)
= (  || use  auxiliary functions f and g:
      (  || defining expression for empty using f and g
      ,  || defining expression for isempty using f and g
      ,  || ---
      :
      ) where  f = .... ; g = .....
  )
```

To be more specific, we shall do the exercise for f and g equal to the identity function, the idea being that f and g play the role of "abs-stack" and "repr-stack" in the analoguous definition in ML. Recall that in ML such functions are required to control the type-checking, in particular the conversion between ~~to and from~~ the abstract type stack * and the concrete type [*]; semantically the two functions are just the identity. Here is the definition.

```
(empty, isempty, push, pop, top)
= ( empty', isempty', push', pop', top')
  where
    f :: stack *  -> [*];
    f x = x
    g :: [*] -> stack *;
    g x = x
```

```
empty'      =  g []
isempty' x  =  f x = []
push' a x   =  g (a: f x)
pop' x      = (g . tl . f) x
top' x      = (hd . f) x
```

Note that the abstype-declaration is left unaffected. In particular the signature stays the same. As the abstype-identifier stack is visible throughout the prevailing scope, it may be used within the definition of (empty, ..., top): see the type declarations for f and g. Note also that the type declarations for f and g may be omitted without disturbing the well-typedness of the script. We have given those types only to show that the ML style of defining an abstract type can be simulated easily in Miranda. (I suspect that the type-checker will find the types * → * for both f and g; nevertheless it also accepts the given types.) Finally note that the definition stack * == [*] should be left unaffected, and cannot be merged into the simultaneous definition of (empty, ..., top).

By now it should be obvious that auxiliary types (be it abstract types, synonyms or algebraic types) can be dealt with in the same way.

## Conclusion

In defining an abstract type in Miranda, auxiliary definitions, which are to be encapsulated completely, are quite well possible. The ML-style of converting between the abstract and concrete type is easily expressed in Miranda -- but is not needed at all--.

All this is true in the supposition that the separate definitions (of the identifiers mentioned in the signature) may be combined into one simultaneous definition (of a tuple containing all of them). I can hardly imagine that this is not the case. (However, if I am wrong, I would suggest to redefine the language!)

## Literature

Bergstra, J.A., Tucker, J.V.: The completeness of the algebraic specification methods for data types. Report IW 156/80, Mathematical Centre (nowadays CWI), Amsterdam, 1980.

Turner, D.A.: Miranda - a non-strict functional Language with polymorphic types. In Functional Programming Languages and Comp. Architecture, (ed J.P. Jouannaud)

LNCS 201 (1985) pp. 1-16.