Locations as first class objects in ML

We describe an extension to the ML type system which (we conjecture) permits the type-secure manipulation of references (i.e. pointers). Before proceeding, note that it's not clear that the unrestricted use of references should be allowed at all (one might argue that our proposal is to letref's as goto's are to while's). Note also that I havn't managed to prove the extended type system secure; indeed I think it's quite possible that it isn't. One purpose of this document is to challenge people to find the holes. My overall goal was to enable polymorphic assignable data-structures (e.g. arrays or lists permitting RPLACA) to be defined as abstract types . To achieve this, together with type security, I've had to introduce a rather ad-hoc notion of "weak polymorphism". This seems to give sufficient polymorphism for the simple examples I've considered, but may turn out to be too constraining. I hope something both more elegant, and more powerful, can eventually be devised.

Weak Polymorphism

We introduce a new kind of type variables: *, **, ***, ...; these will be called weak variables to distinguish them from *, **, ***, ... which we'll call full variables. We require that each argument position of each type operator be specified as either weak or full.

All the argument positions of the standard ML operators are full, however we introduce a new unary operator "ref" whose only argument position is weak. Values of type "ty ref" are references to (i.e. locations holding) values of type ty. To maintain type security each reference must hold values of a fixed monotype - intuitively weak variables stand for undetermined monotypes. To manipulate references we use the following prinitives:

```
newref : * -> * ref
   $:= : * ref # * -> .
  cont : * ref -> *
```

The generic types of these are as shown, their meanings are as follows:

"newref e": e is evaluated and a New location containing e's value is returned.

"e:=e'": e and e' are evaluated in that order; e must evaluate to a location. The evaluation of "e:=e'" has the side effect of storing the value of e' in the value of e; empty is returned.

"cont E": e is evaluated and its value must be a location; the contents of this location is returned.

Note suggesting research topic for Damas, circa 1980?

Typechecking proceeds as in standard ML, together with the following constraints:

1. Types containing full variables may never be substituted for weak variables (any types can be substituted for full variables).

2. For "e e'" to be well-typed, e must have type ty'->ty, e' must have type ty' and all weak variables in ty' must occur in the types of enclosing \-bound varstructs.

3. The ith argument position of a type operator op defined by "abs{rec}type (xl,...,xn)op = ..." is defined to be weak if xi is weak, otherwise it is full.

4. In a type "(tyl,...,tyn)op", if the ith argument position of op is weak then tyi must not contain any full type variables.

To help motivate the first two constraints consider:

```
let f = \x.let r = newref x in (\z.r:=z),(\().cont r)
```

f will be ascribed type: * -> (* -> .) # (. -> *). Now for each application "f e" not occuring inside a \, e must have a definate monotype; thus evaluating "let store,fetch = f([])" at top level is prohibited. However both "let store,fetch = f([]:form)" and "let store,fetch = f([]:thm)" would be allowed.

The third and fourth constraint govern the creation and use of new type operators, for example consider:

```
abstype * array = (* list) # int # int
 with newarray(l,nl,n2) =
       if length l = n2-nl+1
        then absarray(map newref l , nl , n2)
        else failwith `newarray`
 and  select a n =
       let l,nl,n2 = reparray a
       in  if n<nl or n>n2
            then failwith `select`
            else el(n-nl+1)l
             whererec el n l = n=1 => hd l | el(n-1)(tl l)
```

This defines a new unary type operator "array", whose only argument is weak, and whose primitives have generic types:

```
newarray : * list # int # int -> * array
  select : * array -> int -> *
```

To create a new array with lower bound nl, upper bound n2, and initial contents vl,...,vn (where n=n2-nl+1) one evaluates "newarray([vl;...;vn],nl,n2)". To get the value stored at the ith component of a one evaluates "cont(select a i)"; to change the value to v one evaluates "select a i := v".