When Types Get In The Way
A collection of examples

Stefan Kahrs
University of Edinburgh
Laboratory for Foundations of Computer Science

Outline

Type Systems are supposed to help us.

They help us to prevent errors by demanding a form of well-typedness for all parts of a program. For sound type systems this typically means that there is a kind of invariant enforced upon the entire program.

Does this invariant come with a penalty?

Expressibility

There is/could be an expressibility penalty: certain functions may not be expressible at all under the enforced invariant.

```
datatype 'a option = NONE | SOME of 'a
datatype 'a lam =
    Var of 'a | App of 'a lam * 'a lam | Lam of ('a option) lam

fun substitute f (Var x) = f x
 |  substitute f (App(a,b)) = App(substitute f a,substitute f b)
 |  substitute f (Lam t) =
        Lam (substitute (fn NONE => Var NONE |
                            SOME x => lift (f x)) t)
and lift t = substitute (Var o SOME)
```

Performance

Are there other penalties? Can the performance be affected? Can the type system prevent you from writing an efficient version of the function?

In general, the answer is Yes — in particular abstract types show this behaviour.

```
abstype Primrec = Z | S of Primrec
with
    val zero = Z
    val succ = S
    fun recurse mz ms Z = mz
     |  recurse mz ms (S x) = ms (recurse mz ms x)
end
```

And now we want to write...

```
fun pred n = #1 (recurse (zero,zero)
                         (fn(x,y)=>(y,succ y)) n)
```

The function `pred` takes linear time.

Instead, the following does not type-check because of scoping:

```
fun pred Z = Z
 |  pred (S x) = x
```

Apart from that it's observationally equivalent to the previous version.

Polymorphic Recursion

Abstract types are somehow a special case, it's a bit like customizing the type system for your needs. What about other type constructions?

```
datatype 'a mynat = Z | S of 'a mynat mynat
fun succ Z = S Z
 |  succ (S x) = S (succ x)
```

The function `succ` is indistinguishable from `S`, but it has the more general type `'a mynat -> 'b mynat` versus `'a mynat mynat -> 'a mynat`. It is less efficient than `S` but it's impossible to write a $O(1)$ function of the more general type with this behaviour.

Even without polymorphic recursion we have the same problem:

```
datatype 'a mynat' = Z | S of 'a mynat'
```

Beyond Pathological Monsters

Here is an example from SML'90, possibly the most efficient version of list reversal:

```
fun quickreverse l =
    let val r=ref nil
        fun loop [] = !r
         |  loop (x::xs) = (r:= x :: !r; loop xs)
    in  loop l
    end
```

Problem: SML'90 gives this a weak type `'_a list -> '_a list`.
Assuming applicative type variables would have been sound.

Substitution Revisited

We assume polymorphic recursion!
Recall the substitution function for `'a lam`. We can slightly improve on its performance by writing the `lift` function separately:

```
fun substvar f (Var x) = Var (f x)
 |  substvar f (App(a,b)) = App(substvar f a,substvar f b)
 |  substvar f (Lam t) =
        Lam (substvar (fn NONE => NONE |
                          SOME x => SOME (f x)) t)
fun lift t = substvar SOME
```

But that seems to be as far as we can go.

Direct Lift

With de Buijn indices one would normally write a lift function a bit differently, e.g. as follows:

```
fun lift t =
    let fun getvar x 0 = SOME x
          |  getvar NONE y = NONE
          |  getvar (SOME x) y = SOME(getvar x (y-1))
        fun liftn (Var x) n = Var (getvar x n)
          |  liftn (App(t,u)) n = App(liftn t n,liftn u n)
          |  liftn (Lam t) n = Lam (liftn t (n+1))
    in  liftn t 0
    end
```

Doesn't type-check, even under polymorphic recursion.

Substitution Again; No Polyrec

Here is a type for PCF expressions carrying values of a specific (but unspecified) type.

```
datatype ('a,'b) Either = Left of 'a | Right of 'b

datatype 'a Exp =
        App  of 'a Exp * 'a Exp | Lam of 'a Exp |
        Fix of 'a Exp | Zero | Succ of 'a Exp |
        Pred of 'a Exp | If0 of 'a Exp * 'a Exp * 'a Exp |
        Val of (int,'a) Either;
```

What does a substitution function for this type look like?

As we would like it

```
fun subst f (Val n) = f n
|   subst f (App (t,u)) = App (subst f t,subst f u)
|   subst f (Lam t) = Lam (subst (aux f) t)
|   subst f (Fix t) = Fix (subst (aux f) t)
|   subst f Zero = Zero
|   subst f (Succ t) = Succ (subst f t)
|   subst f (Pred t) = Pred (subst f t)
|   subst f (If0(c,t,e)) =
        If0 (subst f c,subst f t,subst f e)
and aux f (Left 0) = Val (Left 0)
|   aux f x = lift(f x)
and lift t = subst (fn Left x => Val(Left(x+1))|
                       Right x=> Val(Right x)) t
```

What happens?

It type-checks, but not with the right type:
we get `((int,'a) Either -> 'a Exp) -> 'a Exp -> 'a Exp`

What we would like is a monad-style type, in which we replace the `'a` by `'b Exp`, i.e. we want type:
`((int,'a) Either -> 'b Exp) -> 'a Exp -> 'b Exp`

Polymorphic recursion finds this type right away.

As we are forced to write it

```
(* repeat previous code and then... *)
fun subst f (Val n) = f n
|   subst f (App (t,u)) = App (subst f t,subst f u)
|   subst f (Lam t) = Lam (subst (aux f) t)
|   subst f (Fix t) = Fix (subst (aux f) t)
|   subst f Zero = Zero
|   subst f (Succ t) = Succ (subst f t)
|   subst f (Pred t) = Pred (subst f t)
|   subst f (If0(c,t,e)) =
        If0 (subst f c,subst f t,subst f e)
and aux f (Left 0) = Val (Left 0)
|   aux f x = lift(f x);
```

An ADS Example

```
fun top2s (op >) [x,y] = ...
 |  top2s (op >) [x,y,z] = ...
 |  top2s (op >) xs =
    let fun lq ((a,_),(b,_)) = a>b
        val (ys,a) = pairing (op >) xs
        val ((m11,m12),(m21,_)) = top2s lq ys
    in  case a of NONE => (m11,max2lq (op >)(m21,m12))
        | SOME a' => if a'>m11 then (a',m11)
                     else (m11,max3lq (op >)(m12,m21,a'))
    end
```

Auxiliary Stuff

```
fun pairing _ [] = ([],NONE)
 |  pairing _ [x] = ([],SOME x)
 |  pairing (op >) (x::y::xs) =
    let val (zs,a)=pairing (op >) xs
    in  if x>y then ((x,y)::zs,a)
        else ((y,x)::zs,a)
    end
```

A teensy weensy example...

```
fun leftfunct f (Left x) = Left (f x)
 |  leftfunct f (Right x) = Right x

fun leftfunct' f (Left x) = Left (f x)
 |  leftfunct' f (r as Right x) = r
```

Now, `leftfunct` has a more general type than `leftfunct'`, but they are observationally equivalent. One could argue that `leftfunct'` is more efficient.

Open Problems

Less pathological examples that move complexity class. Are there any?

With polymorphic recursion, without?

What's the essence of it all? Is there a theory behind this bunch of examples?

Is there a safe area in which we can play this game?
My arguments ultimately refer to the operational semantics, but this can be at odds with what implementations do: consider `pred`, `leftfunct`, possibly the `lift` function with untyped local functions.