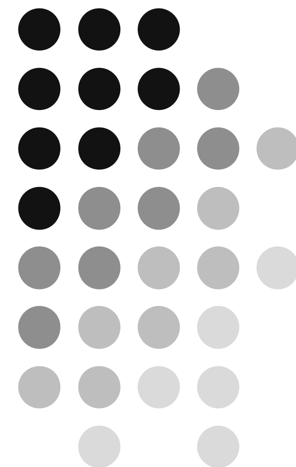
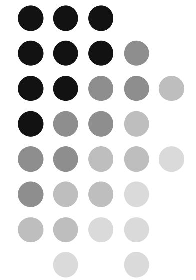
Embedded Interpreters

Nick Benton
Microsoft Research
Cambridge UK



Setting: Scripting languages for SML applications

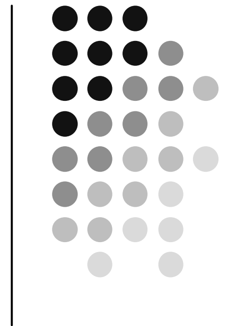


- Application comprises many interesting higher-type values and new type definitions
- Purpose of scripting language (object language) is to give the user a flexible way to glue those bits together at runtime
- Requires more sophisticated interoperability between the two levels than in the self-contained case
- SML tradition is to avoid the problem by not defining an object language at all – just use interactive top-level loop instead.
- Not really viable for stand-alone applications, libraries, interesting object-level syntaxes, situations in which commands come from files, network, etc.

Hal: A Tactical Theorem Prover

The core of this example was written by Larry Paulson and is described in Chapter 10 of his (highly recommended) book ML for the Working Programmer.

```
Hal Applet Compiled by MLj 0.2
goal "(ALL x. P(x)) --> P(a)"
(ALL x. P(x)) --> P(a)
 1. empty  |-  (ALL x. P(x)) --> P(a)
()
by (impR 1)
(ALL x. P(x)) --> P(a)
 1. ALL x. P(x)  |-  P(a)
()
by (allL 1)
(ALL x. P(x)) --> P(a)
 1. P(?_a), ALL x. P(x)  |-  P(a)
()
by (unify 1)
(ALL x. P(x)) --> P(a)
No subgoals left!
()
```



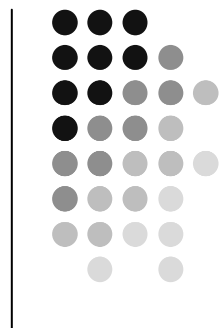
Basic idea

Interpreter uses universal datatype, U :

```
datatype U = UF of U->U | UP of U*U | UUnit |
             UI of int | US of string | UT of tactic
```

Represent types by embedding-projection pairs:

```
type 'a EP
val embed   : 'a EP -> ('a->U)
val project : 'a EP -> (U->'a)

val unit   : unit EP
val int    : int EP
val string : string EP
val **     : ('a EP)*('b EP) -> ('a*'b) EP
val -->    : ('a EP)*('b EP) -> ('a->'b) EP
```



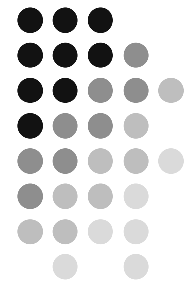
- Use embed to define environment

```
val tacs =
  [("||", embed (tactic**tactic-->tactic) Tacs.||),
   ("repeat", embed (tactic-->tactic) Tacs.repeat),
   ...]
```

- Then can do, for example

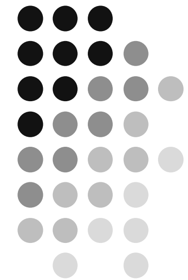
```
interpret (parse “by (repeat (conjR 1))”)
```

- Hooray. What else can we do?



We can embed *and* project

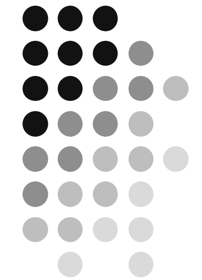
- Hence simple metaprogramming
- Can do polymorphic functions
- …or even untypeable functions
- Can do recursive datatypes in a couple of ways



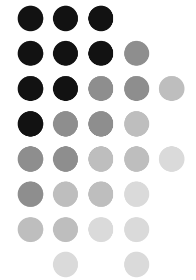
Fun with quote/antiquote:

```
- fun twice f n = f (f n);
- val h =
%%`fn x => ^(embedc ((int-->int)-->int-->int)
             twice) (fn n=>n+1) x`;
val h = fn : staticenv -> dynamicenv -> U
- val hp = projectc (int-->int) h;
val hp = fn : int->int
- hp 2;
val it = 4 : int
```

An amusing factorial function:



```
- let val embY = interpret (read
          "fn f=>(fn g=> f (fn a=> (g g) a))
                 (fn g=> f (fn a=> (g g) a))",[]) []
      val polyY = fn a => fn b=> project
                 (((a-->b)-->a-->b)-->a-->b) embY
      val sillyfact = polyY int int
          (fn f=>fn n=>if n=0 then 1 else n*(f (n-1)))
  in (sillyfact 5) end;

val it = 120 : int
```



Monadic Interpreters

- Would like to parameterize by an arbitrary monad T
- Seems impossible: need an extensional version of CBV monadic translation, which is not definable in core ML or Haskell
- An ML function value of type

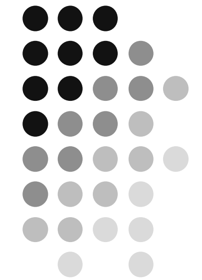
  $(\mathtt{int} \to \mathtt{int}) \to \mathtt{int}$

  needs to be given a semantics in the interpreter of type

  $(\mathtt{int} \to T\,\mathtt{int}) \to T\,\mathtt{int}$

How can the ML function “know what to do” with the extra monadic information returned by calls to its argument?

Filinski to the rescue…



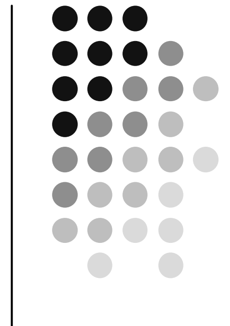
Using first-class control and state, for any monad T *can* define polymorphic functions

```
val reflect : 'a T -> 'a
val reify : (unit -> 'a) -> 'a T
```

This cunning idea combines with representing types by embedding-projection pairs to allow the definition of an *extensional* monadic translation just as we wanted:

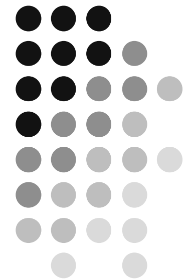
```
type ('a,'astar) TR = ('a->'astar)*('astar->'a)
type 'a BASE = ('a,'a) TR
val int : int BASE

val ** : ('a,'astar) TR * ('b,'bstar) TR ->
              ('a*'b, 'astar*'bstar) TR

val --> : ('a,'astar) TR * ('b,'bstar) TR ->
              ('a->'b, 'astar -> 'bstar R.M.t) TR
```

The embedded monadic interpreter



- Combine the embedding-projection pairs with the monadic translation-untranslation functions
- The monad can be either *implicit* or *explicit* in the universal datatype and the code for the interpreter – we choose implicit, which means no changes to the interpreter itself
- Each type *A* is represented by a 4-tuple
  - $e_A : A \rightarrow U$
  - $p_A : U \rightarrow A$
  - $t_A : A \rightarrow A^*$
  - $n_A : A^* \rightarrow A$
- ML values which represent the operations of the monad will have ML types which are *already* in the image of the (.)* translation
- Embed them by first *untranslating* them, to get an ML value of the type which they will appear to have in the object language and then embedding the result, i.e. $e_A \circ n_A$

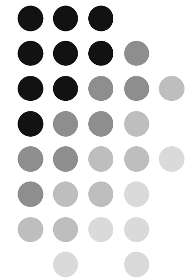
Example: Non-determinism



- Use list monad with monad operations for choice and failure

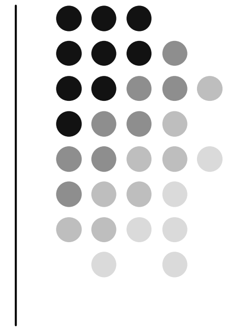
```
fun choose (x,y) = [x,y] (* choose : 'a*'a->'a T *)
fun fail () = []          (* fail : unit->'a T *)

val builtins =
   [("choose", membed (any**any-->any) choose),
    ("fail", membed (unit-->any) fail),
    ("+", embed (int**int-->int) Int.+), ... ]

- project int (interpret (read
   "let val n = (choose(3,4))+(choose(7,9))
    in if n>12 then fail() else 2*n",[])) [];
val it = [20,24,22] : int ListMonad.t
```

Even fancier: π-calculus

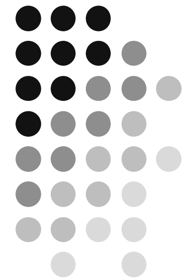


- Well-known translation of CBV λ-calculus into (asynchronous, first-order) π.
- Goal: interpreter for π with embeddings which turn ML functions into processes ,and projections which turn suitably well-behaved processes into ML functions
- Relies on first-class control again. Either
  - Use monadic reflection (“denotational” style)
  - Implement interpreter directly using continuation-based coroutines (Wand, Reppy,…). Imperative style and a bit simpler.

Function case of embedding/projection



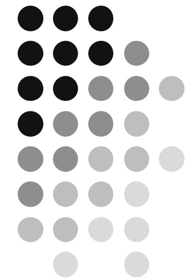
```
fun (ea,pa)-->(eb,pb) =
    (fn f =>
      let val c = new()
          fun action () = let val [ac,VN rc] = receive c
                              val _ = fork action
                              val resc = eb (f (pa ac))
                           in send(rc,[resc])
                           end
       in (fork action; VN c)
       end,
     fn (VN fc) => fn arg =>
        let val ac = ea arg
            val rc = new ()
            val _ = send(fc,[ac,VN rc])
            val [resloc] = receive(rc)
        in pb resloc
        end)
```



Example:

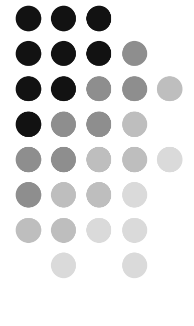
```
val test13 = let val c = ltest "c"
"(new v v!0 | v?*n = c?[x r]=r!n | inc![n v]) |
 (new v v!0 | v?*n = c?[x r]=r!n | inc![n v])"
  in project (unit --> int) c
  end
- test13();
val it = 0 : int
- test13();
val it = 0 : int
- test13();
val it = 1 : int
- test13();
val it = 1 : int
- test13();
val it = 2 : int
```

Summary



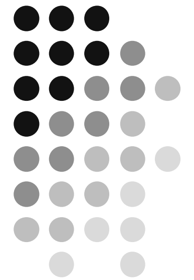
- Embedding higher typed values into lambda calculus interpreter using embedding-projection pairs
- Projecting object-level values back to typed metalanguage
- Polymorphism
- Metaprogramming
- Recursive datatypes
- Embedded monadic interpreter via extensional monadic transform (using monadic reflection and reification)
- Embedded pi-calculus interpreter.

Related work



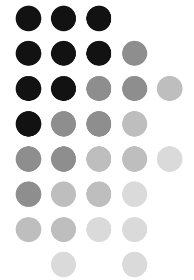
- Modelling types as retracts of a universal domain in denotational semantics
- NbE & TDPE (Berger, Schwichtenberg, Danvy, Filinski, Dybjer, Yang,..)
- printf-like string formatting (Danvy)
- pickling (Kennedy)
- Lua and other extension languages (Ramsey)
- Pict (Turner, Pierce)
- Concurrency and continuations (Wand, Reppy, Claessen,…)

Questions?



http://research.microsoft.com/~nick/

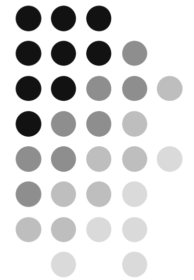
Recursive datatypes



```
datatype U = ... | UT of int*U
val wrap : ('a -> 'b) * ('b -> 'a) -> 'b EP -> 'a EP
val sum  : 'a EP list -> 'a EP
val mu   : ('a EP -> 'a EP) -> 'a EP
```

```
fun wrap (decon,con) ep = ((embed ep) o decon,
                           con o (project ep))
fun sum ss =
   let fun cases brs n x =
         UT(n, embed (hd brs) x)
         handle Match => cases (tl brs) (n+1) x
   in (fn x=> cases ss 0 x,
       fn (UT(n,u)) => project (List.nth(ss,n)) u)
   end
fun mu f = (fn x => embed (f (mu f)) x,
            fn u => project (f (mu f)) u)
```

Usage pattern



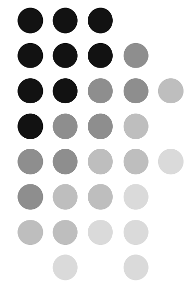
- Given

$$\texttt{datatype d} = C_1 \texttt{ of } \tau_1 \mid \ldots \mid C_n \texttt{ of } \tau_n$$

- The associated EP is

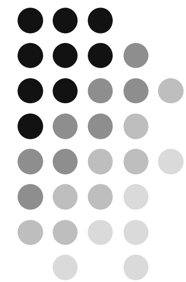
$$\begin{aligned}\texttt{val d = mu (fn z => sum [wrap (fn (}C_1\texttt{ x) => x, }C_1\texttt{)}\ \overline{\tau_1}\texttt{,}\\ \ldots\\ \texttt{wrap (fn (}C_n\texttt{ x) => x, }C_n\texttt{)}\ \overline{\tau_n}\texttt{])}\end{aligned}$$

Example: lists



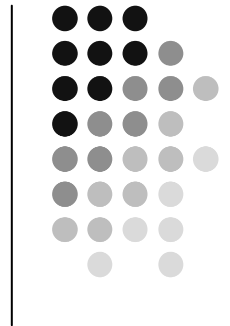
```
- fun list elem = mu ( fn l => (sum
    [wrap (fn []=>(),fn()=>[]) unit,
     wrap (fn (x::xs)=>(x,xs),
           fn (x,xs)=>(x::xs)) (elem ** l)]));
val list : 'a EP -> 'a list EP

(* now extend the environment *)
[...
  ("cons", embed (any**(list any)-->(list any)) (op ::)),
  ("nil", embed (list any) []),
  ("null", embed ((list any)-->bool) null), ... ]
```

Lists continued



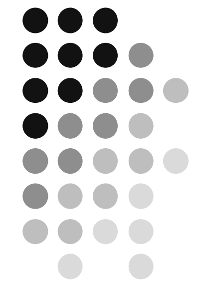
```
- interpret (read
   "let fun map f l = if null l then nil
                      else cons(f (hd l),map f (tl l))
    in map", []) [];
val it = UF fn : U

- project ((int-->int)-->(list int)-->(list int)) it;
val it = fn : (int -> int) -> int list -> int list

- it (fn x=>x*x) [1,2,3];
val it = [1,4,9] : int list
```

That's semantically elegant, but…

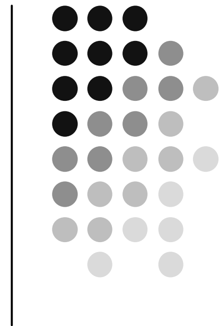


- It's also absurdly inefficient
- Every time a value crosses the boundary between the two languages (twice for each embedded primitive) its entire representation is changed
- Laziness doesn't really help – even in Haskell, that version of map is quadratic
- There is a more efficient approach based on using the extensibility of exceptions to implement a Dynamic type, but
  - It doesn't allow datatypes to be treated polymorphically.
  - If you embed the same type twice, the results are incompatible

Now add variable-binding constructs



```
type staticenv = string list
type dynamicenv = U list
fun indexof (name::names, x) = if x=name then 0 else 1+(indexof(names, x))

(* val interpret : Exp*staticenv -> dynamicenv -> U *)
fun interpret (e,static) = case e of
   EI n => K (UI n)
 | EId s => (let val n = indexof (static,s)
             in fn dynamic => List.nth (dynamic,n)
             end handle Match => let val lib = lookup s builtins
                                 in K lib
                                 end)
 | EApp (e1,e2) => let val s1 = interpret (e1,static)
                       val s2 = interpret (e2,static)
                   in fn dynamic => let val UF(f) = s1 dynamic
                                        val a = s2 dynamic
                                    in f a
                                    end
                   end
 | ELetfun (f,x,e1,e2) =>
                    let val s1 = interpret (e1, x::f::static)
                        val s2 = interpret (e2,f::static)
                        fun g dynamic v = s1 (v::UF(g dynamic)::dynamic)
                    in fn dynamic => s2 (UF(g dynamic)::dynamic)
                    end
```



And it works

```
val test1 =
   "new r1 new r2 twice![inc r1] | r1?f = f![3 r2] | r2?n =
   itos![n echo]"
-test test1;
5

val y = project (((int-->int)-->int-->int)-->int-->int)
(ltest "y" "y?*[f r] = new c new l r!c | f![c l] |
                           l?h = c?*[x r2]= h![x r2]")

val twice = project ((int-->int)-->int-->int)
(ltest "tw" "tw?*[f r] = new c new l r!c |
                c?*[x r2] = f![x l] | l?z = f![z r2]")

val fac = y (fn f=>fn n=>if n = 0 then 1 else n*(f (n-1)))
- twice fac 3;
val it = 720 : int
```