Staging (part II)

March 2018

Last time: staging

.<e>.

This time: more staging

.<e>.

Generalizing **algebraic** optmisation

Staging and **effects**

Review: staging

Multi-stage programming adds support for building code values:

quoting `.<e>.` `.~e` **escaping**

Staging transforms simple functions like `pow` ...

```
let rec pow x n =
  if n = 0 then 1
  else x * pow x (n - 1)
```

... into **code generators**:

```
let rec pow x n =
  if n = 0 then .< 1 >.
  else .< .~x * .~(pow x (n - 1)) >.
```

Review: partially-static structures

A sharp **static**/**dynamic** distinction leads to sub-optimal code:

```
    pow .< x >. 3
⇝
    .< x * x * x * 1 >.
```

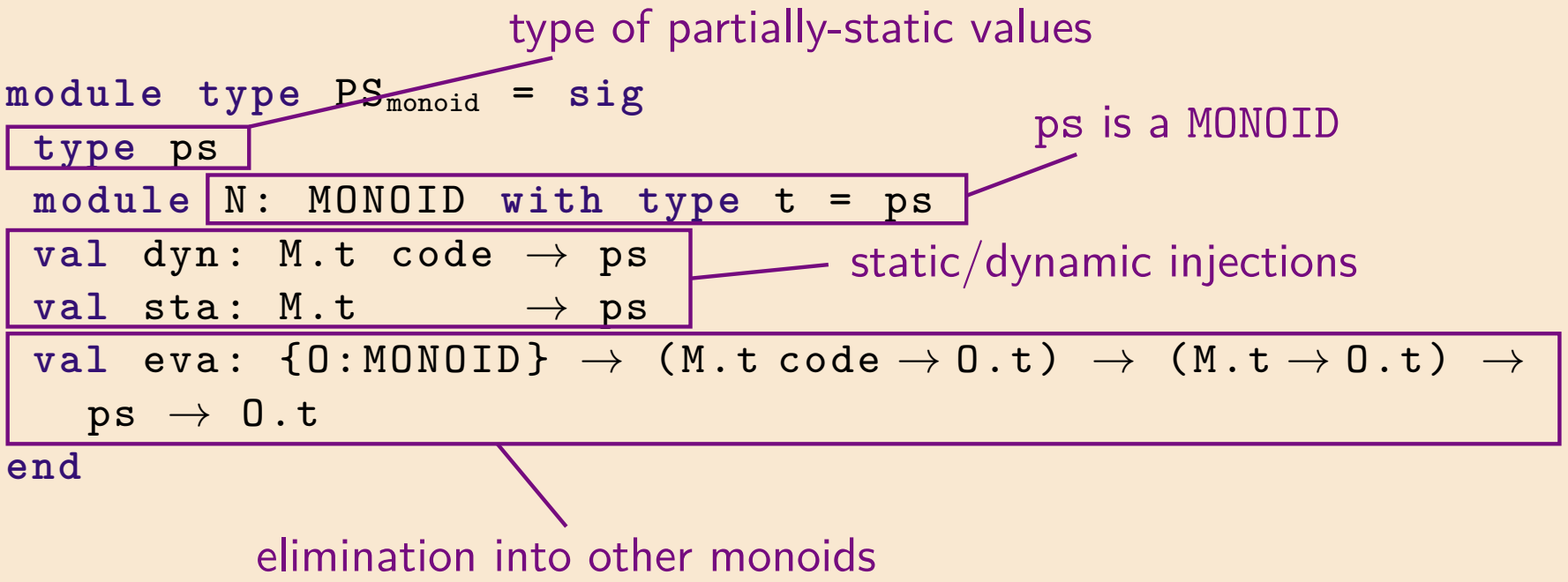
Partially-static structures improve matters using algebraic laws.

Partially-static structures evaluate to **canonical** representations:

```
    (sta 2 <*> dyn .<x>.) <*> (sta 4 <*> dyn .<x>.)
⇝
    (8, {x↦2})
```

Review: partially-static monoid interface

Given a monoid M, define a partially-static monoid type:



Plan: generalize $\mathsf{PS}_{\mathsf{monoid}}$ to arbitrary algebras

Review: partially-static commutative monoids

Partially-static **commutative monoid** representation:

static value & bag of variables ($sx^n y^m \ldots$)

For integers with multiplication:

```
module Ps_cmonoid = struct
 type ps = int * (int IVarMap.t)
 module N = struct type t = ps
                   let unit = (1, IVarMap.empty)
                   let (<*>) (sx,dx) (sy,dy) =
                     (sx*sy, merge add dx dy)
 let sta x = (x, IVarMap.empty)
 let dyn x = (1, IVarMap.singleton x)
 let eva = ...
end
```

(**Note**: extends easily to arbitrary commutative monoids)

Implementing partially-static non-commutative monoids

Monoids are very common. But not all monoids are commutative:

Strings with catenation form a **non-commutative** monoid

```
"a" ^ "b" ≢ "b" ^ "a"
```

Matrices with multiplication form a **non-commutative** monoid

$$\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} 2 & 0 \\ 2 & 2 \end{pmatrix} \not\equiv \begin{pmatrix} 2 & 0 \\ 2 & 2 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$$

The $\texttt{ps}_{\texttt{cmonoid}}$ representation is unsuitable for these monoids.

Generalizing the interface

For an algebraic signature S with an implementation M, define:

```
module type PS_S = sig
 type ps
 module N: S with type t = ps
 val dyn: M.t code → ps
 val sta: M.t      → ps
 val eva: {O:S} → (M.t code → O.t) → (M.t → O.t) →
   ps → O.t
end
```

Examples for S:

monoids, rings, abelian groups, lattices, datatypes ...

Working out the representation

Finding a partially-static representation for an algebra.
Expressions are built using operations, constants and variables

$$(2 \oplus x) \oplus (4 \oplus y)$$

Next, consider **equivalence classes** of expressions under the laws.
For monoids, reassociated expressions are equivalent:

$$(2 \oplus x) \oplus (4 \oplus y)$$
$$((2 \oplus x) \oplus 4) \oplus y$$
$$2 \oplus ((x \oplus 4) \oplus y)$$
$$\ldots$$

Finally, pick a **canonical representative** of the equivalence class.
Every monoid expression can be fully left- (or right-) associated

$$2 \oplus (x \oplus (4 \oplus y))$$

Working out the representation (continued)

Adjacent constants can be combined. For example, for strings:

```
x ^ ("a" ^ ("b" ^ y))
  ≡
x ^ ("ab" ^ y)
```

So, the partially-static representation for monoids:

a sequence with **no adjacent static elements**

For **commutative monoids** the order of elements is irrelevant.

Constants can be grouped together and reduced.

So, the partially-static representation for commutative monoids:

a pair of a **single static** element and a **bag of variables**

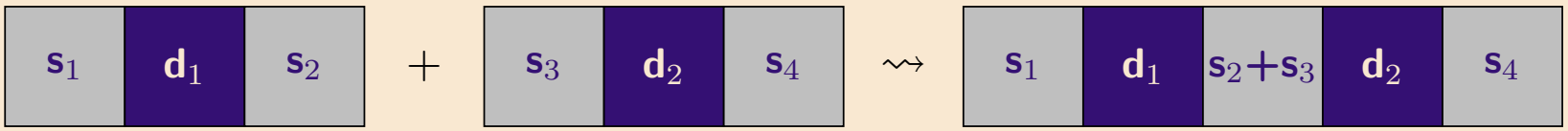
The (non-commutative) monoid representation

Sequence with no adjacent **static** elements.

```
type sta = Sta and dyn = Dyn and eseq = E: _ seq → eseq
and _ seq =  (* sta seq begins with a static element *)
    Empty : _ seq
  | ConsS : string       * dyn seq → sta seq
  | ConsD : string code *  _   seq → dyn seq

let consS : string → eseq → eseq =
  fun x xs → match xs with
    | E Empty → E (ConsS (x, Empty))
    | E (ConsS (y, ys)) → E (ConsS (x ^ y, ys))
    | E (ConsD (_,_) as r) → E (ConsS (x, r))

let rec (<*>) : eseq → eseq → eseq = ...
```



Example: typed format specifiers

Typed format strings (for `printf`/`scanf`) as a GADT:

```
type (_,_) fmt =
  | Int : (int → 'a, 'a) fmt
  | Lit : string → ('a, 'a) fmt
  | Bool : (bool → 'a, 'a) fmt
  | Cat : ('a,'b) fmt * ('b,'c) fmt → ('a,'c) fmt
let (%) x y = Cat (x, y)
```

Used in the standard library (over 6000 lines of formatting code!)

The **first parameter accumulates** `printf`/`scanf` argument types:

```
Lit "(" % Int % Lit "," % Int % Lit ")"
  : (int → int → 'a, 'a)
```

Example: typed printf (unstaged)

Typed printf (in **continuation-passing style**):

```
let rec printk : type a r. (a, r) fmt → (string → r
  fun fmt k → match fmt with
  | Int → fun i → k (string_of_int i)
  | Bool → fun b → k (string_of_bool b)
  | Lit s → k s
  | Cat (l, r) → printk l (fun x →
                   printk r (fun y →
                   k (x ^ y)))
```

A top-level function for **printing to strings**:

```
let sprintf : type a. (a, string) fmt → a =
  fun fmt → printk fmt (fun x → x)
```

`sprintf` in action:

```
sprintf (Bool % Lit "," % Lit " " % Bool %) true false
  ⇝ "true, false"
```

Example: printf, naively staged

Format strings are **static**
printf **arguments** and **accumulator** are **dynamic**:

```
let rec printk2
 : type a r. (a,r) fmt → (string code → r code) → a code
= fun fmt k → match fmt with
   | Int → .< fun i → .~(k .<string_of_int i>.) >.
   | Bool → .< fun b → .~(k .<string_of_bool b>.) >.
   | Lit s → k .< s >. (* CSP *)
   | Cat (l, r) → printk2 l (fun x →
                  printk2 r (fun y →
                  k .< .~x ^ .~y >.))

let sprintf2 : type a. (a, string) fmt → a code =
 fun fmt → printk2 fmt (fun x → x)
```

Staged printf output

```
sprintf2 (Bool % Lit "," % Lit " " % Bool %)
⇝
.< fun b_1 b_2 →
    ((string_of_bool b_1 ^ ",") ^ " ") ^ string_of_bool b_2>.
```

Plan: reduce catenation count using partially-static monoids

Staged printf with partially-static monoids

Improve binding-times using `dyn` and `sta`:

```
let rec printk3
 : type a r. (a, r) fmt → (Ps_string.t → r code) → a code
  = fun fmt k → match fmt with
  | Int → .< fun i → .~(k (dyn .<string_of_int i>.)) >.
  | Bool → .< fun b → .~(k (dyn .<string_of_bool b>.)) >.
  | Lit s → k (sta s)   (* better! *)
  | Cat (l, r) → printk3 l (fun x →
                  printk3 r (fun y →
                  k (x <*> y)))

let sprintf3 : type a. (a, string) fmt → a code =
  fun fmt → printk3 fmt Ps_string.cd
```

Staged printf with partially-static monoids: output

```
sprintf3 (Bool % Lit "," % Lit " " % Bool %)
⇝
 .< fun b_1 b_2  →
     string_of_bool b_1 ^ (", " ^ string_of_bool b_2)>.
```

Static strings catenated **during generation**, improving the output.

Plan: reduce catenations further using effects.

Example: inner product

Specification:

$$\texttt{dot n [|}x_1\texttt{; }x_2\texttt{; ...; }x_n\texttt{|] [|}y_1\texttt{; }y_2\texttt{; ...; }y_n\texttt{|]}$$

$$\rightsquigarrow \texttt{(}x_1 \times y_1\texttt{) + (}x_2 \times y_2\texttt{) + ... + (}x_n \times y_n\texttt{)}$$

Implementation:

```
let dot
 : int → float array → float array → float
 = fun n l r →
    let rec loop i =
      if i = n then 0.
      else l.(i) *. r.(i)
        +. loop (i + 1)
    in loop 0
```

Binding-time analysis

Classify variables into **dynamic** (`'a code`) / **static** (`'a`)

```
let dot
 : int → float array code → float array code → float code
  = fun n l r →
```

dynamic: `l`, `r`
static: `n`

Classify expressions into static (no dynamic variables) / dynamic

```
if i = n then 0
else l.(i) *. r.(i)
```

dynamic: `l.(i) *. r.(i)`
static: `i = n`

Goal: reduce static expressions during code generation.

Inner product, loop unrolling

Length-specialized `dot`:

```
let dot'
: int → float array code → float array code → float code
 = fun n l r →
    let rec loop i =
      if i = n then .< 0. >.
      else .< ((.~l).(i) *. (.~r).(i))
                +. .~(loop (i + 1)) >.
    in loop 0
```

As with `pow`, making the loop variable static unrolls the loop:

```
# .< fun l r → .~(dot' 3 .< l >. .< r >.) >.;;
- : (float array → float array → float) code =
.< fun l r →
    (l.(0) *. r.(0)) +.
      ((l.(1) *. r.(1)) +. ((l.(2) *. r.(2)) +. 0.))>.
```

Inner product, argument specialization

Vector-specialized dot:

```
let dot"
: int → float array → float array code → float code
 = fun n l r →
    let rec loop i =
      if i = n then .< 0. >.
      else let li = l.(i) in
      .< (li *. (.~r).(i)) +. .~(loop (i + 1)) >.
    in loop 0
```

Generated code:

```
.< fun a → .~(dot" 3 [| 0.0; 1.0; 2.0|] .<a>.)>.
⇝
.< fun a  → (0. *. a.(0)) +.
            (1. *. a.(1)) +.
            (2. *. a.(2)) +. 0.))>.
```

Partially-static algebra for inner product

Float addition and multiplication form a **commutative semiring**.

The partially-static representation is a **polynomial**:

$$s_1 x^{i_1} y^{j_1} + s_2 x^{i_2} y^{j_2} + \ldots + s_m x^{i_m} y^{j_m}$$

(As usual we can define a **generic** commutative semiring structure)

Inner product, carefully staged

The "staged" code now needs **no staging annotations**:

```
let dot {R:RING}
 : int → R.t array → R.t array → R.t
 = fun n l r →
    let rec loop i =
      if i = n then R.zero
      else l.(i) * r.(i) + loop (i + 1)
    in loop 0
```

Algebraic simpliciation improves the output:

```
.< fun a →
  .~(dot 3 [|sta 0.0; sta 1.0; sta 2.0|]
           [| dyn .<a.(0)>.;
              dyn .<a.(1)>.;
              dyn .<a.(2)>.|])>.
⇝
.< fun a  → a.(1) +. (2. *. a.(2)).)>.
```

Metaprogramming and effects

Generative metaprogramming and compositionality

Naive generative metaprogramming is **compositional**.

The code generated by an expression is the **composition** of the code generated by subexpressions

```
f .<x>.⇝ .< ... x ... >.
```

Generativity & compositionality has useful **safety properties**:
— no way to violate scoping in generate code

However, it is difficult to generate **optimal code** this way.

Generative metaprogramming, compositionality and effects

Consequence of compositionality:
generated code structure follows generator structure

Benefit: no arbitrary reordering of code

Drawback: need to write generator in awkward style (e.g. CPS)

`let` insertion: interface

Plan: add operations for inserting `let` at a **higher level** but at an outer level

Interface:

```
(* Mark the place where let can be inserted *)
val let_locus : (unit -> 'a code) -> 'a code

(* Insert a let binding where let_locus was called;
   return the variable *)
val genlet : 'a code -> 'a code
```

Example:

```
let_locus (fun () -> .< 1 + .~(genlet .<f y>.) >.
 ⇝
   .< let x = f y in 1 + x >.
```

let insertion: a simple implementation

Implementation of `let` insertion:

```
effect GenLet : 'a code -> 'a code
let genlet v = perform (GenLet v)

let let_locus : (unit → 'a code) → 'a code =
  fun f → match f () with
  | x → x
  | effect (GenLet e) k →
    .< let x = .~e in .~(continue k .< x >. )>.
```

if-insertion is also useful. (Need to run the continuation **twice**).

```
effect Split : bool code → bool
let split b = perform (Split b)

let if_locus f = match f () with
  | x → x
  | effect (Split b) k →
      .< if .~b then .~(continue k true)
         else .~(continue (Obj.clone k) false) >.
```

Now we can turn a dynamic bool into two static bools

```
.< fun b → .~(if_locus @@ fun () →
                cd (sta "b:" <*>
                    string_of_bool (split b))) >.
⇝
.< fun b → if b then "b:true"
                else "b:false" >.
```

Improving printf with `let` & `if` insertion

```
let rec printk4
  : type a r. (a, r) fmt → (Ps_string.t → r code) → a code =
  fun fmt k → match fmt with
  | Int → genlet
          .< fun i → .~(k (dyn .<string_of_int i>.)) >.
  | Bool → genlet
           .< fun b →
            .~(if_locus (fun  () →
               k (sta (string_of_bool (split .<b>.))))) >.
  | Lit s → k (sta s)
  | Cat (l, r) → printk4 l (fun x →
                  printk4 r (fun y →
                  k (x <*> y)))
```

Note: `b` is dynamic, but `string_of_bool` builds a **static value**!

Improving printf with `let` insertion: output

```
sprintf4 (Bool % Lit "," % Lit " " % Bool %)
⇝
  .< let f b = if b then "(false,true)"
               else "(false,false)" in
     let g b = if b then "(true,true)"
               else "(true,false)" in
     let h b = if b then g else f in h >.
```

No catenations!

Pair splitting

One more effect: `genlet2` inserts **pair bindings**.

```
effect Genlet2 : ('a * 'b) code → 'a code * 'b code
let genlet2 p = perform (Genlet2 p)
```

A **handler** for `genlet2`:

```
let let2_locus f =
  match f () with
  | x → x
  | effect (Genlet2 p) k →
    .< let (x,y) = .~p in .~(continue k (.<x>.,.<y>.))>.
```

`genlet2` in action:

```
.< fun p → .~(let2_locus @@ fun () →
                let (a, b) = genlet2 .<p>. in
                (.< ((.~a, .~b), .~a) >.)) >.
⇝
.< fun p → let (x, y) = p in ((x, y), x) >.
```

Final example: unstaged scanf

The `sscanf` function reads from a string according to a format:

```
let rec sscanfk
  : type a r. (a, r) fmt → string → a → r * string =
  fun fmt s k → match fmt with
    | Int →          let i, s = read_int s in
                     (k i, s)
    | Lit l →        let (), s = read_exact l s in
                     (k, s)
    | Bool →         let i, s = read_bool s in
                     (k i, s)
    | Cat (l, r) → let k1, s = sscanfk l s k in
                     let k2, s = sscanfk r s k1 in
                     (k2, s)

let sscanf : 'a 'b. ('a, 'b) fmt → string → 'a → 'b =
  fun f s k → fst (sscanfk f s k)
```

`sscanf` in action

```
sscanf (Bool % Lit ", " % Bool) "true, false" (fun x y -> (x,y))
 ⇝ (true, false)
```

Naively-staged scanf

Staged `sscanfk2` after naive staging:

```
let rec sscanfk2
  : type a r. (a, r) fmt → string code → a code
      → (r * string) code =
  fun fmt s k → match fmt with
    | Int →       .< let i, s = read_int .~s in
                     (.~k i, s) >.
    | Lit l →     .< let (), s = read_exact l .~s in
                     (.~k, s) >.
    | Bool →      .< let i, s = read_bool .~s in
                     (.~k i, s) >.
    | Cat (l, r) → .< let k1, s = .~(sscanfk2 l s k) in
                       let k2, s = .~(sscanfk2 r .<s>. .<k1>.)
                       in (k2, s) >.

let sscanf2 : 'a 'b. ('a, 'b) fmt → (string → 'a → 'b) code =
  fun f → .< fun s k → fst .~(sscanfk2 f .<s>. .<k>.) >.
```

Naively-staged scanf: output

```
sscanf2 (Bool % Lit ", " % Bool)
⇝
.< fun s1 k1  ->
   fst (let (k2,s2) =
          let (k3,s3) =
           let (i1,s4) = read_bool s1 in (k1 i1, s4) in
          let (k4,s5) =
           let ((),s5) = read_exact ", " s3 in (k3, s5) in
          (k4, s5) in
         let (k5,s6) =
          let (i2,s7) = read_bool s2 in (k2 i2, s7) in
         (k5, s6))>.
```

Plan: remove administrative bindings using `genlet2`

Staged scanf with genlet2

```
let rec sscanfk3
  : type a r. (a, r) fmt → string code → a code → r code * strin
  fun fmt s k → match fmt with
    | Int →          let i, s = genlet2 .<read_int .~s>. in
                     (.<.~k .~i>., s)
    | Lit l →        let (), s = genlet2 .<read_exact l .~s>. in
                     (k, s)
    | Bool →         let i, s = genlet2 .<read_bool .~s>. in
                     (.<.~k .~i>., s)
    | Cat (l, r) → let k1, s = .~(sscanfk3 l s k) in
                     let k3, s = .~(sscanfk3 r s k1) in
                     (k3, s)

let sscanf3 : 'a 'b. ('a, 'b) fmt → (string → 'a → 'b) code =
  fun f → .< fun s k → .~(let_locus (fun () →
                              fst (sscanfk3 f .<s>. .<k>.))) >.
```

Staged scanf with genlet2: output

```
sscanf3 (Bool % Lit ", " % Bool)
⇝
. < fun s k ->
      let (x1 ,s1) = read_bool s
      let (x2 ,s2) = read_exact "," s1 in
      let (x3 ,s3) = read_bool s2
        k x 2 >
```

No administrative bindings!

Summary

Partially-static structures for algebraic **simplification**

(`2 + x + 4`)

Algebraic **effects** for code **motion**

(`genlet .<e>.`)

Aim: move work from **generated** code to **generator**

Next time: recursion

$\mu \mathtt{x}\,.\,\mathtt{e}$