A monadic approach for avoiding code duplication when staging memoized functions

Kedar Swadi, Walid Taha,

Oleg Kiselyov, Emir Pasalic

Rice University and FNMOC

Background

- Code duplication is a problem whenever we generate code (FFTW, Fridge, Stratego, …)
- Slows down both
  - Generation time, and
  - Runtime of generated program
- Particular domain: Dynamic programming
  - Interesting feature: Memoization
  - Widely used (potential killer app)

Ideal: Code would look like math

```
// gib : int * (int * int) -> int
let rec gib (n,(x,y)) =
  match n with
  | 0 -> x
  | 1 -> y
  | _ -> gib (n-2,(x,y)) +
         gib (n-1,(x,y))
```

Reality: Need memoization

Let’s use a simple implementation of a store:

```
let s0 = fun x -> None  // empty store
let ext s n m =         // extend
    fun y -> if n = y
             then Some m
             else s y
let lookup s n  = s n   // lookup
```

Memoizing code

```
// gib_memo : int * (int * int) -> state -> state * int
let rec gib_memo (n,(x,y)) s =
  match (lookup s n) with
   | Some r -> (s,r)
   | None ->
       (match n with
       | 0 -> (s,x)
       | 1 -> (s,y)
       | _ ->
         let (s1,r1) = gib_memo (n-2,(x,y)) s in
         let (s2,r2) = gib_memo (n-1,(x,y)) s1 in
         (ext s2 n (r1+r2), r1+r2))
```

Great! Now let’s stage it!

```
// gib_memo'  : int * (int code * int code)
             -> state' -> state' * int code
let rec gib_memo' (n,(x,y)) s =
  match (lookup s n) with
  | Some r -> (s,r)
  | None ->
      (match n with
      | 0 ->  (s,x)
      | 1 ->  (s,y)
      | _ ->
         let (s1,r1) = gib_memo' (n-2,(x,y)) s in
         let (s2,r2) = gib_memo' (n-1,(x,y)) s1 in
         (ext s2 n .< .~r1 + .~r2 >. , .< .~r1 + .~r2 >.))
```

How well does this work?

It's a valid staging

- MSP talk: It's a well-typed two-level program
- PE talk: Congruence is satisfied

Is it useful?

```
.< fun (x,y) ->
   .~(snd (gib_memo' (5,(.<x>., .<y>.))
                          s0)) >.;;
```

Returns:

```
.<fun (x, y) ->
   ((y + (x+y)) + ((x+y) + (y + (x+y))))>.
```

What's the source of this problem?

```
// gib_memo'  : int * (int code * int code)
             -> state' -> state' * int code
let rec gib_memo' (n,(x,y)) s =
  match (lookup s n) with
  | Some r -> (s,r)
  | None ->
      (match n with
      | 0 ->  (s,x)
      | 1 ->  (s,y)
      | _ ->
         let (s1,r1) = gib_memo' (n-2,(x,y)) s in
         let (s2,r2) = gib_memo' (n-1,(x,y)) s1 in
         (ext s2 n .< .~r1 + .~r2 >. , .< .~r1 + .~r2 >.))
```

This example illustrates that

the code duplication problem arises when

- we have partially dynamic data
- when components of this structure need to share code

affects both specialization time and runtime

should happen almost all the time

How do we usually deal with duplication?

The power function can help us explain key points

```
// square : int -> int
let square x = x * x

// power : int -> int -> int
let rec power x n =
  if n = 0 then 1
  else if n mod 2 = 0
  then square (power x (n / 2))
  else x * (power x (n -  1 ))
```

A first attempt at staging it

```
// square' :  int code -> int code
let square' x = .< .~x * .~x >.

// power' : int code -> int -> int code
let rec power' x n =
  if n = 0 then .<1>.
  else if n mod 2 = 0
  then square' (power' x (n/2))
  else .< .~x * .~(power' x (n-1)) >.

.< fun x -> .~(power' .<x>. 5) >.;;
```

Returns: `.<fun x -> (x * (((x*1) * (x*1)) * ((x*1) * (x*1))))>.`

Easily fixed: “Let-insertion”

```
// square' :  int code -> int code
let square' x = .< let y = .~x  in y * y >.

// power' : int code -> int -> int code
let rec power' x n =
  if n = 0 then .<1>.
  else if n mod 2 = 0
  then square' (power' x (n/2))
  else .< .~x * .~(power' x (n-1)) >.

.< fun x -> .~ (power' .<x>. 3) >.;;
```

Returns:
```
.<fun x -> (x * let y_3  = let y_2  = (x * 1)
                                   in (y_2 * y_2)
                        in (y_3 * y_3))>.
```

Does it always work?

Consider a more “complex” power:

```
// ( ** ) : int * int -> int * int -> int * int
let ( ** ) (a,b) (c, d) = (a*c - b*d, a * d + b * c)

let square x = x ** x

let rec power x n =
  if n = 0 then (1,0)
  else if n mod 2 = 0
  then square (power x (n / 2))
  else x ** (power x (n -  1 ))
```

Let’s stage it

```
// ( ** ) : (int code * int code) -> …same… -> …same…
let ( ** ) (a,b) (c,d) =
      (.< .~a * .~c - .~b * .~d >. , .< .~a * .~d + .~b * .~c >.)

// square'  : int code * int code -> int code * int code
let square' x = x ** x

// power' : int code * int code -> int -> int code * int code
let rec power' x n =
  if n = 0 then ( .<1>., .<0>.)
  else if n mod 2 = 0
  then square' (power' x (n/2))
  else x ** (power' x (n-1))
```

Does this work?

This is exactly the same problem with gib

Solution: Convert to CPS

```
// ( ** ) : int code * int code -> int code * int code
        -> (int code * int code -> 'a     ) -> 'a

let ( ** ) (a,b) (c,d) = fun k ->




        k ( .< .~a * .~c - .~b * .~d >.,
             .< .~a * .~d + .~b * .~c >.)
```

What CPS lets us do

```
// ( ** ) : int code * int code -> int code * int code
         -> (int code * int code -> 'a code) -> 'a code

let ( ** ) (a,b) (c,d) = fun k ->
  .< let a' = .~a  in
     let b' = .~b in
     let c' = .~c in
     let d' = .~d in
     .~(k ( .< a' * c' - b' * d' >.,
             .< a' * d' + b' * c' >.)) >.
```

Solution: Convert to CPS

Rest is easy

```
let square' x = x ** x

let rec power' x n k =
  if n = 0 then k ( .< 1 >., .< 0 >.)
  else if n mod 2 = 0
  then (power' x (n/2)) (fun r -> square' r k)
  else (power' x (n-1)) (fun r -> (r ** x) k)
```

Back to gib: Convert to CPS

```
// gib_memo' : int * (int code * int code)
            -> state' -> (state' * (int code) -> 'a) -> 'a

let rec gib_memo' (n,(x,y)) s k  =
  match (lookup s n) with
  | Some r -> k (s,r)
  | None ->
       (match n with
       | 0 -> k (s,x)
       | 1 -> k (s,y)
       | _ ->
        gib_memo' (n-2,(x,y)) s (fun (s1,r1) ->
        gib_memo' (n-1,(x,y)) s1 (fun (s2,r2) ->
        k (ext s2 n .< .~r1 + .~r2 >. , .< .~r1 + .~r2 >.))))
```

Recall source of duplication...

```
// gib_memo' : int * (int code * int code)
            -> state' -> (state' * (int code) -> 'a) -> 'a

let rec gib_memo' (n,(x,y)) s k  =
  match (lookup s n) with
  | Some r -> k (s,r)
  | None ->
       (match n with
       | 0 -> k (s,x)
       | 1 -> k (s,y)
       | _ ->
        gib_memo' (n-2,(x,y)) s (fun (s1,r1) ->
        gib_memo' (n-1,(x,y)) s1 (fun (s2,r2) ->
        k (ext s2 n .< .~r1 + .~r2 >. , .< .~r1 + .~r2 >.))))
```

Now we can insert a let-statement

```
// gib_memo' : int * (int code *  int code)
             -> state' -> (state' * int code -> 'ans code) -> 'ans code

let rec gib_memo' (n,(x,y)) s k  =
  match (lookup s n) with
  | Some r -> k (s,r)
  | None ->
       (match n with
       | 0 -> k (s,x)
       | 1 -> k (s,y)
       | _ ->
        gib_memo' (n-2,(x,y)) s (fun (s1,r1) ->
        gib_memo' (n-1,(x,y)) s1 (fun (s2,r2) ->
        .< let z = .~r1 + .~r2
           in .~(k (ext s2 n .<z>., .<z>.)) >.)))
```

Code we generate now

```
.<fun (x,y) -> .~(gib_memo' (5, (.<x>., .<y>.)) s0
                                  fun (_, b) -> b )>. ;;
```

Returns `.<fun (x, y) ->`

```
            let z_3  = x + y in
            let z_4  = y + z_3 in
            let z_5  = z_3 + z_4 in
            let z_6  = z_4 + z_5
            in z_6>.
```

No more code duplication. We won! But…

Is this still like the math?

```
// gib_memo' : int * (int code *  int code)
            -> state' -> (state' * int code -> 'ans code) -> 'ans code

let rec gib_memo' (n,(x,y)) s k  =
  match (lookup s n) with
  | Some r -> k (s,r)
  | None ->
       (match n with
       | 0 -> k (s,x)
       | 1 -> k (s,y)
       | _ ->
        gib_memo' (n-2,(x,y)) s (fun (s1,r1) ->
        gib_memo' (n-1,(x,y)) s1 (fun (s2,r2) ->
        .< let z = .~r1 + .~r2
           in .~(k (ext s2 n .<z>., .<z>.)) >.)))
```

Our contribution

```
let gib_ms f (n, (x, y)) =
match n with
| 0 -> ret x
| 1 -> ret y
| _ ->
   bind (f ((n-2), (x, y))) (fun r1 ->
   bind (f ((n-1), (x, y))) (fun r2 ->
   ret .<.~r2 + .~r1>.))
```

Our contribution

Programmer

- Converts program into monadic form (std)
- Stages this program as usual

One set of monadic combinators

- Encapsulates memoization
- Encapsulates dealing with code duplication

Works for standard DP algorithms

- HMM, knapsack, LCS, matrix mult, OBST
- Generate implementations generally much faster (2-90x) than C implementations (bottom-up, matrix based)

Related Work

Moggi 1990, Wadler 1991, many others

Consel and Danvy 1991, Bondorf 1992, others

- Nobody says “CPS helps let-insertion”

Hatcliff and Danvy 1997

Liu and Stoller 2003, Acar et all 2003

Thiemann 2003

Paper discusses several other related works

Summary

Code duplication makes writing any program generator (or transformation) challenging

Some duplication problems are easier to deal with than others

The difficult cases will come up often

CPS helps a bit. Monads help a lot! Paper

- Explains how monads help in more detail
- Reports on experiments to check these claims

Future Work

Past Future

- Used for generating FFT circuits (EMSOFT 2004, ICESS 2004)
- Used for generating implementations of Gaussian Elimination (Carette and Kiselyov @ GPCE 2005)

Present Future

- Controlled unfolding
  - specialization-free partial evaluation