

Computational Effects across Generated Binders

Part 1: Problems and solutions

Yukiyoshi Kameyama

University of Tsukuba

Oleg Kiselyov

Chung-chieh Shan

TBD

IFIP WG2.11

September 5, 2011

Outline

► Problems

Requirements for the solution

Solutions



Our Power. Our Future.

Power

```
val power : int → int → int  
let rec power n x = match n with  
  | 0 → 1  
  | n → x * power (n-1) x
```

Power

```
val spower : int → ('a, int) code → ('a, int) code  
let rec spower n x = match n with  
  | 0 → ⟨1⟩  
  | n → ⟨~x * ~(spower (n-1) x)⟩
```

Power

```
val spower : int → ('a, int) code → ('a, int) code
```

```
let rec spower n x = match n with
```

```
| 0 → ⟨1⟩
```

```
| n → ⟨~x * ~(spower (n-1) x)⟩
```

```
let spowern n = ⟨fun x → ~(spower n ⟨x⟩)⟩
```

```
spowern 5;;
```

```
↪ - : ('a, int → int) code =
```

```
⟨fun x_1 → (x_1 * (x_1 * (x_1 * (x_1 * (x_1 * 1))))))⟩
```

Power

```
val spower : int → ('a, int) code → ('a, int) code
```

```
let rec spower n x = match n with
```

```
| 0 → ⟨1⟩
```

```
| n → ⟨~x * ~(spower (n-1) x)⟩
```

```
let spowern n = ⟨fun x → ~(spower n ⟨x⟩)⟩
```

```
spowern 5;;
```

```
↪ - : ('a, int → int) code =
```

```
⟨fun x_1 → (x_1 * (x_1 * (x_1 * (x_1 * (x_1 * 1))))))⟩
```

```
spowern (-1);;
```

```
↪ Stack overflow during evaluation (looping recursion?).
```

Faulty Power

Code generation with exceptions

```
exception BadArg
```

```
let rec spowerE n x = match n with
```

```
| 0 → ⟨1⟩
```

```
| n when n > 0 → ⟨ $\sim x * \sim(\text{spowerE } (n-1) x)$ ⟩
```

```
| - → raise BadArg
```


Faulty Power

Code generation with exceptions

```
exception BadArg
let rec spowerE n x = match n with
  | 0 → ⟨1⟩
  | n when n > 0 → ⟨~x * ~(spowerE (n-1) x)⟩
  | _ → raise BadArg
```

```
let spowernE n = ⟨fun x → ~(spowerE n ⟨x⟩) ⟩
```

```
let rec gpower () =
  print_endline "Enter_n:_";
  let n = read_int () in
  try spowernE n
  with BadArg →
    print_endline "Bad_n!";
    gpower ()
```

Faulty Power

Code generation with exceptions

```
exception BadArg
let rec spowerE n x = match n with
  | 0 → ⟨1⟩
  | n when n > 0 → ⟨~x * ~(spowerE (n-1) x)⟩
  | - → raise BadArg
```

```
let spowernE n = ⟨fun x → ~(spowerE n ⟨x⟩) ⟩
```

```
let rec gpower () =
  print_endline "Enter_n:_";
  let n = read_int () in
  try spowernE n
  with BadArg →
    print_endline "Bad_n!";
    gpower ()
```

Guard Insertion

The need to move open code

$\langle \mathbf{fun} \ y \rightarrow \sim \text{complex_code} + 10 / y \rangle$

Guard Insertion

The need to move open code

```
let guarded_div x y =  
  ⟨(assert (~y ≠ 0); ~x / ~y)⟩ in  
  
  ⟨fun y → ~complex_code + ~(guarded_div ⟨10⟩ ⟨y⟩ )⟩
```

```
↪ - : ('a, int → int) code =  
⟨fun y_15 → the_complex_code +  
  begin assert (y_15 ≠ 0); (10 / y_15) end⟩
```

Guard Insertion

Now really moving open code, across binders

```
⟨fun y →  
  ~ (new_ctx (fun () →  
    ⟨~ complex_code + ~(guarded_div 0 ⟨10⟩ ⟨y⟩ )⟩ ) )⟩
```

```
↪ ⟨fun y_4 →  
  assert (y_4 ≠ 0);  
  the_complex_code + 10 / y_4⟩
```

Guard Insertion

Now really moving open code, across binders

```
⟨fun y → ~ (new_ctx (fun () →  
  ⟨fun x → ~ (new_ctx (fun () →  
    ⟨~ complex_code + ~ (guarded_div 1 ⟨x⟩ ⟨y⟩ )) )) ))⟩
```

```
↪ ⟨fun y_5 →  
  assert (y_5 ≠ 0);  
  fun x_6 → the_complex_code + x_6 / y_5⟩
```

Guard Insertion

Now really moving open code, across binders

```
⟨ fun y → ∼(new_ctx (fun () →  
  ⟨(fun x → ∼(new_ctx (fun () →  
    ⟨∼complex_code + ∼(guarded_div 1 ⟨x⟩ ⟨y⟩ )⟩ )))  
    (∼complex_code + ∼(guarded_div 0 ⟨5⟩ ⟨y-1⟩)))⟩ )⟩ )⟩
```

```
↪ ⟨ fun y_7 →  
  assert (y_7 ≠ 0); assert ((y_7 - 1) ≠ 0);  
  ((fun x_8 → the_complex_code + x_8 / y_7  
    (the_complex_code + 5 / (y_7 - 1))))⟩
```

Loop Tiling

Introduction

$$v'_i = \sum_j a_{ij} v_j$$

```
let mvmul0 n m a vin vout =  
  Array.fill vout 0 n 0;  
  for i = 0 to n-1 do  
    for j = 0 to m-1 do  
      vout.(i) ← vout.(i) + a.(i).(j) * vin.(j)  
  done done
```


Loop Tiling

Introduction

$$v'_i = \sum_j a_{ij} v_j$$

```
let mvmul1 b n m a vin vout =  
  Array.fill vout 0 n 0;  
  sloop 0 (n-1) b (fun ii →  
    sloop 0 (m-1) b (fun jj →  
      for i = ii to min (ii + b - 1) (n - 1) do  
        for j = jj to min (jj + b - 1) (m - 1) do  
          vout.(i) ← vout.(i) + a.(i).(j) * vin.(j)  
        done done));
```

Loop Tiling Puzzle

Moving open code with binders across binders

```
let gmvmul1 loop1 loop2 n m = ⟨fun a vin vout →  
  Array.fill vout 0 n 0;  
  ~ (loop1 0 (n-1) (fun i →  
    loop2 0 (m-1) (fun j →  
      ⟨vout.(~i) ← vout.(~i) + a.(~i).(~j) * vin.(~j)⟩ )))  
  )
```

Loop Tiling Puzzle

Moving open code with binders across binders

```
let gmvmul1 loop1 loop2 n m = ⟨fun a vin vout →  
  Array.fill vout 0 n 0;  
  ~ (loop1 0 (n-1) (fun i →  
    loop2 0 (m-1) (fun j →  
      ⟨vout.(~i) ← vout.(~i) + a.(~i).(~j) * vin.(~j)⟩ )))  
  )
```

```
let gen_regular_loop lb ub body =  
  ⟨for i = lb to ub do ~ (body ⟨i⟩ ) done⟩
```

```
gmvmul1 gen_regular_loop gen_regular_loop 10 5  
↪ ⟨fun a_14 vin_15 vout_16 → Array.fill vout_16 0 10 0;  
  for i_17 = 0 to 9 do  
    for i_18 = 0 to 4 do ...
```

Loop Tiling Puzzle

Moving open code with binders across binders

```
let gen_nested_loop b lb ub body =  
  ⟨sloop lb ub b (fun ii →  
    for i = ii to min (ii + b - 1) ub do ~(body ⟨i⟩) done)⟩
```

```
gmvmul1 (gen_nested_loop 2) (gen_nested_loop 2) 10 5  
↪ fun a_19 vin_20 vout_21 → Array.fill vout_21 0 10 0;  
  for ii_22 = 0 to 9 step 2 do  
    for i_23 = ii_22 to min ((ii_22 + 2) - 1) 9 do  
      for jj_24 = 0 to 4 step 2 do  
        for j_25 = jj_24 to min ((jj_24 + 2) - 1) 4 do ...
```

Loop Tiling Puzzle

Moving open code with binders across binders

```
gmvmul1 (gen_nested_loop 2) (gen_nested_loop 2) 10 5  
↪ <fun a_19 vin_20 vout_21 → Array.fill vout_21 0 10 0;  
    for ii_22 = 0 to 9 step 2 do  
      for i_23 = ii_22 to min ((ii_22 + 2) - 1) 9 do  
        for jj_24 = 0 to 4 step 2 do  
          for j_25 = jj_24 to min ((jj_24 + 2) - 1) 4 do ...
```

```
let p = new_prompt () in  
    gmvmul1 (insert_here p (gen_tile_loop p 2))  
            (gen_tile_loop p 2) 10 5  
↪ <fun a_26 vin_27 vout_28 → Array.fill vout_28 0 10 0;  
    for ii_29 = 0 to 9 step 2 do  
      for jj_31 = 0 to 4 step 2 do  
        for i_30 = ii_29 to min ((ii_29 + 2) - 1) 9 do  
          for j_32 = jj_31 to min ((jj_31 + 2) - 1) 4 do ...
```

Outline

Problems

▶ **Requirements for the solution**

Solutions

Scope extrusion

We want to move open code, but not too far

```
let r = ref ⟨0⟩ in  
  ⟨fun y → ~ (r := ⟨y⟩; ⟨1⟩) )⟩ ;  
!r
```

↪ - : ('a, int) code = ⟨y_6⟩

Scope extrusion

We want to move open code, but not too far

```
⟨ fun y → ~ (new_ctx ( fun () →  
  ⟨ fun x → ~ (new_ctx ( fun () →  
    ⟨ ~complex_code + ~ (guarded_div 1 ⟨y⟩ ⟨x⟩ )) )) ) ) )
```

```
↪ ⟨ fun y_34 →  
  assert (x_35 ≠ 0);  
  fun x_35 → the_complex_code + (y_34 / x_35) ⟩
```


Unacceptable

- ▶ Tree hacking

Unacceptable

- ▶ Tree hacking
- ▶ Need to look at the generated code

Unacceptable

- ▶ Tree hacking
- ▶ Need to look at the generated code
- ▶ Post-validation

Unacceptable

- ▶ Tree hacking
- ▶ Need to look at the generated code
- ▶ Post-validation
- ▶ Treating the generated code as white-box

The Goal

Generate code

- ▶ with compositional combinators
- ▶ statically assure well-formed and well-typed code
- ▶ even for the intermediate, open results

CPS/monadic style no longer helps

Simple let-insertion

let genlet e k = $\langle \mathbf{let} \ t = \sim e \ \mathbf{in} \ \sim(k \ \langle t \rangle) \ \rangle$

genlet e1 (**fun** t1 \rightarrow ... genlet e2 k)

\rightsquigarrow $\langle \mathbf{let} \ t1 = \sim e1 \ \mathbf{in} \ \dots \ \mathbf{let} \ t2 = \sim e2 \ \dots \ \rangle$

Inner genlet, inner let-expression

Even nested CPS cannot insert let beyond the closest binding

because abstractions are always pure values

We need a new CPS hierarchy

Outline

Problems

Requirements for the solution

▶ **Solutions**

MetaHaskell

Matrix-vector multiplication, textbook

$$v'_i = \sum_j a_{ij} v_j$$

```
mvmul0 n m a vin vout =  
  clear_vec (int n) vout ;  
  loop_ (int 0) (int (n-1)) (int 1) (lam $ \i →  
    loop_ (int 0) (int (m-1)) (int 1) (lam $ \j →  
      (vec_set ◇ weakens vout ◇ weakens (var i)) ⊙  
      (vec_get ◇ weakens vout ◇ weakens (var i)) ⊕  
      (mat_get ◇ weakens a ◇ weakens (var i) ◇ var j) ⊗  
      (vec_get ◇ weakens vin ◇ var j)  
    ))
```


MetaHaskell

Matrix-matrix addition, tiled

```
mvmul2 b n m a vin vout =  
  clear_vec (int n) vout ;  
  (resetJ $  
  loop_nested_exch b 0 (n-1) (lam $ \i →  
    loop_nested_exch b 0 (m-1) (lam $ \j →  
      (vec_set ◇ weakens vout ◇ weakens (var i)) ⊙  
      (vec_get ◇ weakens vout ◇ weakens (var i)) ⊕  
      (mat_get ◇ weakens a ◇ weakens (var i) ◇ var j) ⊗  
      (vec_get ◇ weakens vin ◇ var j)  
    )))
```

MetaHaskell

Matrix-matrix addition, tiled

```
mvmul2 b n m a vin vout =  
  clear_vec (int n) vout ;  
  (resetJ $  
  loop_nested_exch b 0 (n-1) (lam $ \i →  
    loop_nested_exch b 0 (m-1) (lam $ \j →  
      (vec_set ◇ weakens vout ◇ weakens (var i)) ⊙  
      (vec_get ◇ weakens vout ◇ weakens (var i)) ⊕  
      (mat_get ◇ weakens a ◇ weakens (var i) ◇ var j) ⊗  
      (vec_get ◇ weakens vin ◇ var j)  
    )))
```

MetaHaskell

Loop combinators

Strip-mining

```
loop_nested b lb ub body =  
  loop_ (int lb) (int ub) (int b) (lam $ \ii →  
    loop_ (var ii) (min_ (var ii) +: int (b-1)) (int ub)) (int 1)  
    (weakens body))
```

Tiling: strip-mining + exchange

```
loop_nested_exch b lb ub body =  
  let_ (insloop (int lb) (int ub) (int b)) (\ii →  
    loop_ (var ii) (min_ (var ii) +: int (b-1)) (int ub)) (int 1)  
    (weakens body))
```

PEPM 2012

ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation

January 23-24, 2012

Philadelphia, Pennsylvania, USA

co-located with POPL'12

- ▶ Submission deadline: October 3
- ▶ Notifications: November 8

<http://www.program-transformation.org/PEPM12>

Conclusions

Effectful code generation

- ▶ Effects are desirable to write good-looking generators
- ▶ Effects are necessary for loop tiling, loop-invariant code motion, assertion-insertion and the movement of open code across binders

Prototype of MetaHaskell

- ▶ Like MetaOCaml: generation of assuredly well-typed and well-scoped code
- ▶ Unlike MetaOCaml: safety guarantees in the presence of *arbitrary* effects