Computational Effects across Generated Binders
Part 1: Problems and solutions

Yukiyoshi Kameyama Oleg Kiselyov Chung-chieh Shan
University of Tsukuba TBD

IFIP WG2.11
September 5, 2011

Code generation is the leading approach to making high-performance software reusable. Using a set of realistic examples, we demonstrate that side effects are indispensable in composable code generators, especially side effects that move open code past generated binders. We challenge the audience to implement these examples in their favorite code-generation framework.

We implemented the examples ourselves using a prototype library of code-generating combinators in Haskell. This library statically assures not only that all generated code is well-formed and well-typed but also that all generated variables are bound lexically as expected. Such assurances are crucial for code generators to be written by domain experts rather than compiler writers, because the most profitable optimizations are domain-specific ones.

Outline

- **Problems**

  Requirements for the solution

  Solutions

The talk is to motivate code generation with effects. We will present several motivating examples, which are all based on real code. Please regard them as challenges. You are welcome to try to solve them in your favorite code generation framework. Not just any solution will do however! We will talk about the requirements a bit later in the talk. We will concentrate on effects that, informally, cross the future-stage binders. Our first example will illustrate what that means.

The first example is simplistic, as befits the first example. Its goal to familiarize ourselves with the notation and terminology.

Power

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

Here is the standard power function, raising `x` to the power `n`.

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)⟩
```

This is the staged power, to generate code to raise a statically unknown quantity to the statically known power. Here `n` is still `int` but the variable `x` has the type `('a, int) code`: the code that will produce at run-time an `int` value. This is also the type of `spower`. Here `.< >.` are 'brackets' (think of List quasi-quote) and $\sim$ is the 'escape' or splice (think of Lisp anti-quotation).

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))))) ⟩
```

We are not done yet: whence `x` will come from? We need another definition: `spowern` to specialize power to the statically-known exponent. Here is the example: The loop in `spower` is executed at the generation time; the result is the fully unrolled code.

We see in `spowern` the binding of a future-stage variable; That variable can be turned into a piece of code `.<x>.` and manipulated by the generator (spliced into code templates). The future-stage variable is being manipulated 'symbolically', so to speak.

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 ?).
```

The power function is such a cliche! It so easy to overlook the problem: the function is partial. I submit that overflowing the stack (which may lead to the segmentation fault on some platforms) is not the friendliest way to report problems.

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
```

We wish to throw exceptions and being able to recover from them.
We coin a new benchmark: Faulty Power.

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 ()
```

The function `gpower` demonstrates interactive code generation with exception handling. We ask the user for the exponent and generate the specialized power. If the code generator `spowernE` throws an exception, the user is scolded and asked to re-enter the exponent.

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 ()
```

We stress: the exception is thrown in `spowerE` and is caught in `gpower`. In-between there is a future-stage binder, in `spowernE`. The simplest, cliche example of staging already demonstrates the need for effects, which cross future-stage binders.

If we write this in Haskell and use the Error monad, we see the problem right away: the escape requires a code value but `spowerE n .<x>.` is a computation.

Guard Insertion

The need to move open code

$$\langle \textbf{fun}\ \mathsf{y} \rightarrow \sim \mathsf{complex\_code} + 10\ /\ \mathsf{y} \rangle$$

The previous example was very simple: an effect (exception) was indeed propagating through the future-stage binder, but no values were carried along in the exception. Such effects, involving values of base types (not code and not functions!) are unproblematic and can easily be accommodated in existing frameworks, perhaps after ad hoc extensions. Mint can do it, and so can our PEPM09 calculus. Now we show how to move *open* code. Take a look at the expression on the slide, which could be the output of a program generator. Division is a partial operation. We wish to assure that it always succeeds, so we insert the run-time y non-zero test.

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⟩
```

We re-write our expression using a generator for guarded division, which assures that the divisor is non-zero. In the first approximation, `guarded_div` could be defined as shown on the slide. The result is not satisfactory: if `y` does turn out zero, we waste time computing the complex expression before throwing an exception. One would say that the new code is not better than the partial division. Run-time assurance tests ought to be executed as soon as possible, to avoid wasting time computing the results that would be thrown away. In our case, the assert has to be *moved* right after the binder, which is the earliest possible moment to test `y`. We are moving the open code!

Guard Insertion

Now really moving open code, across binders

$$\langle \textbf{fun}\ \mathsf{y} \rightarrow \sim(\mathsf{new\_ctx}\ (\textbf{fun}\ ()\ \rightarrow \langle \sim\mathsf{complex\_code} + \sim(\mathsf{guarded\_div}\ 0\ \langle 10\rangle\ \ \langle \mathsf{y}\rangle\ )\rangle\ ))\rangle$$

$$\rightsquigarrow\ \langle \textbf{fun}\ \mathsf{y\_4} \rightarrow \textbf{assert}\ (\mathsf{y\_4} \neq 0); \mathsf{the\_complex\_code} + 10\ /\ \mathsf{y\_4}\rangle$$

Here is what we want: we write the generator as shown on the slide, and get the result like the one below. The non-zero assertion is done right after the binder, where we want it. (The mysterious first argument `0` of `guarded_div` is to be discussed later.) We have moved the open code across the addition expression. But we want more.

Guard Insertion

Now really moving open code, across binders

$$\langle \textbf{fun}\ \mathsf{y} \rightarrow {\sim}(\mathsf{new\_ctx}\ (\textbf{fun}\ ()\ \rightarrow \langle \textbf{fun}\ \mathsf{x} \rightarrow {\sim}(\mathsf{new\_ctx}\ (\textbf{fun}\ ()\ \rightarrow \langle {\sim}\mathsf{complex\_code} + {\sim}(\mathsf{guarded\_div}\ 1\ \langle \mathsf{x} \rangle\ \ \langle \mathsf{y} \rangle\ )\rangle\ ))\rangle\ ))\rangle$$

$$\rightsquigarrow\ \ \langle \textbf{fun}\ \mathsf{y\_5} \rightarrow \textbf{assert}\ (\mathsf{y\_5} \neq 0); \mathsf{fun\ x\_6} \rightarrow \mathsf{the\_complex\_code} + \mathsf{x\_6}\ /\ \mathsf{y\_5}\rangle$$

Here we generate code with two binders, `x` and `y`. The test for `y` being non-zero should be performed as early as possible, that is, right after the `y` binder. Thus we have to move open code across future-stage binders (the binder for `x`, in our case).

Guard Insertion

Now really moving open code, across binders

$$
\begin{array}{l}
\langle \textbf{fun}\ \mathsf{y} \rightarrow \sim(\mathsf{new\_ctx}\ (\textbf{fun}\ ()\ \rightarrow \\
\quad \langle (\textbf{fun}\ \mathsf{x} \rightarrow \sim(\mathsf{new\_ctx}\ (\textbf{fun}\ ()\ \rightarrow \\
\qquad \langle \sim \mathsf{complex\_code} + \sim(\mathsf{guarded\_div}\ 1\ \langle \mathsf{x} \rangle\ \ \langle \mathsf{y} \rangle\ )\rangle\ ))) \\
\quad (\sim \mathsf{complex\_code} + \sim(\mathsf{guarded\_div}\ 0\ \langle 5 \rangle\ \ \langle \mathsf{y}{-}1 \rangle))\rangle\ ))\rangle \\
\\
\rightsquigarrow\ \ \langle \textbf{fun}\ \mathsf{y\_7} \rightarrow \\
\quad \textbf{assert}\ (\mathsf{y\_7} \neq 0);\ \textbf{assert}\ ((\mathsf{y\_7} - 1) \neq 0); \\
\quad ((\textbf{fun}\ \mathsf{x\_8} \rightarrow \mathsf{the\_complex\_code} + \mathsf{x\_8}\ /\ \mathsf{y\_7}) \\
\quad (\mathsf{the\_complex\_code} + 5\ /\ (\mathsf{y\_7} - 1)))\rangle
\end{array}
$$

We can even move from several places, from several contexts. Here, one guarded division is under the `x` binder (inside the function) and the other is in the argument expression to which the function applies. The assertions move up and collect under the binder. One can imagine general constraint posting and solving.

One has probably noticed a bit of scaffolding, like mysterious first argument to `guarded_div`. It betrays the real solution. Apparently we should know the nesting level of an expression, or the length of its future-stage environment. MetaOCaml can give us that information, in principle. In our Haskell solution, this information is available for free. We also get the hint that we need a more involved combinator to create binders.

Loop Tiling

Introduction

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

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

The most advanced is loop tiling, which we describe on the example of vector-matrix multiplication: here how it looks in Math and in code. The matrix `a` has `n` rows and `m` columns; `v` is the input vector and `v'` is the output one. We assume that the input vector `v` is long, that is `m` is much greater than `n`.

Loop Tiling

Introduction

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

```
let mvmul1 b n m a v v' =
  Array.fill v' 0 n 0;
  sloop 0 (m−1) b (fun jj →
    sloop 0 (n−1) b (fun ii →
      for j = jj to min (jj + b−1) (m−1) do
        for i = ii to min (ii + b−1) (n−1) do
          v'.( i ) ← v'.( i ) + a.( i ).( j ) ∗ v.( j )
      done done))
```

Here is the same matrix-vector multiplication implemented in tiled loops, with the square tile of size `b`. The function `sloop` is the `for`-loop with the step (step `b` in our case). We see that tiling is converting a single loop into a nested loop and `exchange`, pulling the `ii` loop right after the `jj` loop. The body of the loops remains exactly the same as before; it is executed the same number of times – but in a different pattern.

The array `v` is traversed repeatedly. By assumption, it is long and so won't fit in cache. A tiled program deals with the array a chunk (of size `b`) at a time. A `jj`-th chunk will be loaded into cache, used several times. When we are finished with the chunk, it won't be needed again and can safely be replaced in cache with another chunk. Tiling improves locality taking advantage of cache, and is one of the basic optimizations in high-performance computing. Obviously tiling is not a general-purpose optimization: our loop rearrangement heavily relied on the fact evaluations of loop bodies are uncorrelated.

The tiled code looks more complex; it seems quite easy to make a mistake when tiling by hand. We need automation. We need automation even more when we will be combining loop tiling with scalar promotion, partial unrolling and other optimizations.

Loop Tiling Puzzle

Moving open code with binders across binders

```
let gmvmul1 loop1 loop2 n m = ⟨fun a v v' →
  Array.fill v' 0 n 0;
 ∼(loop1 0 (m−1) (fun j →
     loop2 0 (n−1) (fun i →
     ⟨v'.( ∼i ) ← v'.( ∼i ) + a.(∼i).( ∼j ) ∗ v.( ∼j )⟩ )))
    ⟩
```

Here is the puzzle to build ordinary and tiled loop nests by composition. For example, we may write the vector-matrix multiplication as shown. The code is the straightforward staging of the naive computation, assuming statically known dimensions. The code is 'obviously' correct. We have abstracted the loop as combinators.

Loop Tiling Puzzle

Moving open code with binders across binders

```
let gmvmul1 loop1 loop2 n m = ⟨fun a v v' →
  Array.fill v' 0 n 0;
 ∼(loop1 0 (m−1) (fun j →
     loop2 0 (n−1) (fun i →
     ⟨v'.( ∼i ) ← v'.( ∼i ) + a.(∼i).( ∼j ) ∗ v.( ∼j )⟩ )))
    ⟩

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

gmvmul1 gen_regular_loop gen_regular_loop 5 10
⇝ ⟨fun a_1 v_2 v' _3 → Array.fill v' _3 0 5 0;
    for j_4 = 0 to 9 do
     for i_5 = 0 to 4 do ...
```

If we instantiate both loop arguments as `gen_regular_loop`, we get the naive code seen earlier, with two nested loops.

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) 5 10
⇝ ⟨fun a_19 v_20 v' _21 → Array.fill v' _21 0 5 0;
    for jj_22 = 0 to 9 step 2 do
      for j_23 = jj_22 to min ((jj_22 + 2) − 1) 9 do
        for ii_24 = 0 to 4 step 2 do
          for i_25 = ii_24 to min ((ii_24 + 2) − 1) 4 do ...
```

Choosing a different combinator, we split each loop with the factor of 2 (so-called strip mining). (We have abused the notation and introduced the for-loop with a step. In OCaml, it is implemented as a combinator.)

Loop Tiling Puzzle

Moving open code with binders across binders

```
gmvmul1 (gen_nested_loop 2) (gen_nested_loop 2) 5 10
⇝ ⟨fun a_19 v_20 v' _21 → Array.fill v' _21 0 5 0;
      for jj_22 = 0 to 9 step 2 do
       for j_23 = jj_22 to min ((jj_22 + 2) − 1) 9 do
         for ii_24 = 0 to 4 step 2 do
          for i_25 = ii_24 to min ((ii_24 + 2) − 1) 4 do ...

let p = new_prompt () in
  gmvmul1 (insert_here p (gen_tile_loop p 2))
          (gen_tile_loop p 2) 5 10
⇝ ⟨fun a_26 v_27 v' _28 → Array.fill v' _28 0 5 0;
     for jj_29 = 0 to 9 step 2 do
       for ii_31 = 0 to 4 step 2 do
        for j_30 = jj_29 to min ((jj_29 + 2) − 1) 9 do
          for i_32 = ii_31 to min ((ii_31 + 2) − 1) 4 do ...
```

Finally, if we instantiate the loop argument differently still, we exchange two loops and obtain the tiled code that we have seen on the previous slide. We write loop body once, and apply various transformations (strip-mining, unrolling, etc) many times. In particular, we exchange loop bodies, moving open code with binders across other binders. (Exercise to the reader: what happens if we move `insert_here` over `gmvmul1`?)

Outline

Problems

- **Requirements for the solution**

Solutions

You might have noticed that all the shown problem examples had an implementation. I indeed have the solution for all these examples in MetaOCaml. I do not like it.

Scope extrusion

We want to move open code, but not too far

$$\textbf{let}\ \ \mathsf{r}\ =\ \textbf{ref}\ \langle 0\rangle\ \ \ \textbf{in}$$
$$\ \langle \textbf{fun}\ \mathsf{y}\ \rightarrow\ \sim(\mathsf{r}\ :=\ \ \langle \mathsf{y}\rangle;\ \ \langle 1\rangle\ \ )\rangle\ \ ;$$
$$\ \ !\,\mathsf{r}$$

$$\rightsquigarrow\quad -\ :\ (\,'\,\mathsf{a},\ \ \mathsf{int}\,)\ \ \mathsf{code}\ =\ \langle \mathsf{y\_6}\rangle$$

Effects crossing binders a liable to cause scope extrusion. Here is an example: the mutation effect (similar to the one we've seen earlier) crosses the binder. The result is the code with an unbound variable. We do not wish that to happen! The generated code should be assuredly well-formed.

Scope extrusion

We want to move open code, but not too far

$$
\begin{array}{l}
\langle \textbf{fun}\ \mathsf{y} \rightarrow \sim(\mathsf{new\_ctx}\ (\textbf{fun}\ ()\ \rightarrow \\
\quad \langle \textbf{fun}\ \mathsf{x} \rightarrow \sim(\mathsf{new\_ctx}\ (\textbf{fun}\ ()\ \rightarrow \\
\qquad \langle \sim \mathsf{complex\_code} + \sim(\mathsf{guarded\_div}\ 1\ \langle \mathsf{y} \rangle\ \ \langle \mathsf{x} \rangle\ )\rangle\ ))\rangle\ ))\rangle \\
\\
\rightsquigarrow \quad \langle \textbf{fun}\ \mathsf{y\_34} \rightarrow \\
\qquad \textbf{assert}\ (\mathsf{x\_35} \neq 0); \\
\qquad\ \textbf{fun}\ \mathsf{x\_35} \rightarrow \mathsf{the\_complex\_code} + (\mathsf{y\_34}\ /\ \mathsf{x\_35})\rangle
\end{array}
$$

The problem is real. For example, if we make a simple mistake in the assertion-insertion code, we may generate the following code. Anyone can tell what is wrong with it?

Unacceptable

- Tree hacking

Tree hacking is the term from linguistics, which means rewriting ASTs as first-order data structures (treating the generated code as *free* term algebra.) We do not reject tree hacking: it is fully appropriate and even necessary in compiler construction, when used by an expert (who will then prove the code generator correct: an example is the work of Thiemann and Dussart, 1996-1999). Assembly is a good language, say, for embedded systems, when written by an expert and verified/proved. Most programmers should avoid assembly, and tree hacking. This is especially true for people who build applications and solve 'real' problems, rather than write programming tools.

Unacceptable

- Tree hacking
- Need to look at the generated code

Users of a code generator may not even know the target language. Even if they do, they may have trouble understanding it since the generated code is often too large, too complex, too obfuscated. Therefore, we require that the generated code compile without errors.

Unacceptable

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

One way to assure that the generated code is well-formed is to try to compile it at the end, rejecting it if the compilation fails. Template Haskell uses this approach. Alas, the errors, if any, will be reported too late and in terms of the generated code. We want errors to be reported early, and in terms of the generators. We want the generated code to be well-formed and well-typed all the time, as it is being generated.

The post-validation does *not* help with accidental variable capture errors.

Unacceptable

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

We pursue the pure generative approach: the generated code is treated as black-box and cannot be examined. The generative approach has the strongest equational theory. (For the challenge, we might relax this criterion and allow looking at the generated code, provided that well-formedness and well-typedness are still statically ensured, at all times.)

The Goal

Generate code

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

Our goal is to generate code with compositional combinators that statically assure the results (even intermediate, open results) are well-formed and well-typed.

CPS/monadic style no longer helps

Simple let-insertion

let genlet e k = ⟨**let** t = ∼e **in** ∼(k ⟨t⟩)⟩

genlet e1 (**fun** t1 → ... genlet e2 k)
⇝ ⟨**let** t1 = ∼e1 **in** ... **let** t2 = ∼e2 ... ⟩

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

As it was known in Partial Evaluation community, we can use CPS/monadic style to insert let. We easily see that nesting `genlet` leads to the corresponding nesting of let-statements. We cannot insert let beyond the closest binder! Even nesting of CPS transform does not help. Hint: in the ordinary CPS hierarchy, abstractions are pure values. We need a new CPS hierarchy.

Outline

Problems

Requirements for the solution

- **Solutions**

The code shown in the section about problems was in MetaOCaml. We can indeed solve all the posed problems. Alas, the MetaOCaml solution was not safe: scope extrusion was easily possible. Now, we show the safe code, using safe Template Haskell.

MetaHaskell

Matrix-vector multiplication, textbook

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

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

This is the code using our library, safe code generator embedded in Haskell. The syntax could be better, there is lots of room to design nicer-looking combinators.

MetaHaskell

Matrix-matrix addition, tiled

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

Here is the same matrix-matrix addition with the tiled loops. The loop bodies remain the same, the loop combinator is different. The function mvmul2, depending on the type instantiation, could either give the code to run at the present stage, or give the code to print (and run later). The produced tiled code is the same as we have seen earlier (modulo the syntactic differences between Haskell and OCaml).

MetaHaskell

Matrix-matrix addition, tiled

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

The function resetJ marks the spot where to move the loops.

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))
```

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