

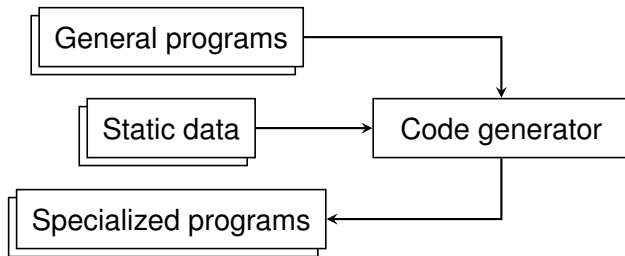
Shifting the stage

Staging with delimited control

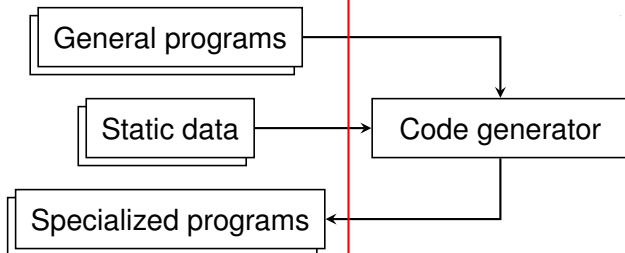
Yukiyoshi Kameyama	Oleg Kiselyov	Chung-chieh Shan
University of Tsukuba	FNMOC	Rutgers
kameyama@acm.org	oleg@pobox.com	ccshan@rutgers.edu

PEPM, 20 January 2009

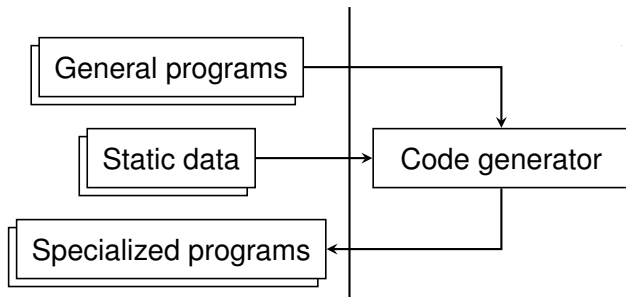
The need for domain-specific code generators



The need for domain-specific code generators



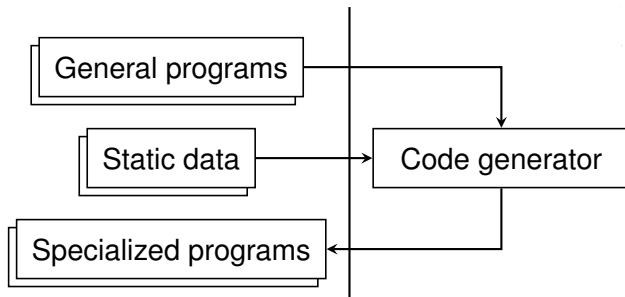
The need for domain-specific code generators



Optimizations specific to ...

- ▶ Gaussian elimination
- ▶ Fast Fourier Transform
- ▶ Linear signal processing
- ▶ Embedded devices

The need for domain-specific code generators



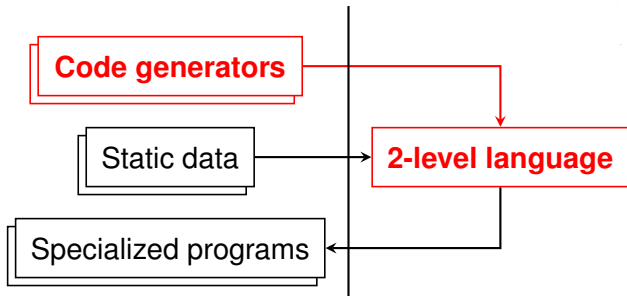
Optimizations specific to ...

- ▶ Gaussian elimination
- ▶ Fast Fourier Transform
- ▶ Linear signal processing
- ▶ Embedded devices

Generate code using ...

- ▶ Binding-time annotations
- ▶ Extensible compilers
- ▶ Side effects
- ▶ Custom generators

The need for domain-specific code generators



Optimizations specific to ...

- ▶ Gaussian elimination
- ▶ Fast Fourier Transform
- ▶ Linear signal processing
- ▶ Embedded devices

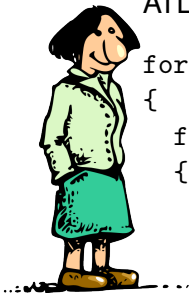
Generate code using ...

- ▶ Binding-time annotations
- ▶ Extensible compilers
- ▶ Side effects
- ▶ **Custom generators**

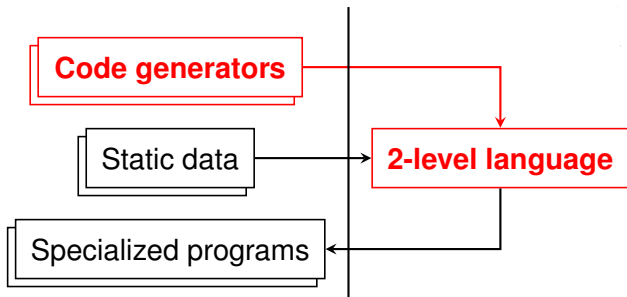
The need for domain-specific code generators

ATLAS generates optimized code for matrix multiplication:

```
for (j=0; j < nu; j++)
{
    for (i=0; i < mu; i++)
    {
        if (Asg1stC && !k)
            fprintf(fpout, "%s  %s%d_%d = %s%d * %s%d;\n",
                    spc, rC, i, j, rA, i, rB, j);
        else
            fprintf(fpout, "%s  %s%d_%d += %s%d * %s%d;\n",
                    spc, rC, i, j, rA, i, rB, j);
        opfetch(fpout, spc, nfetch, rA, rB, pA, pB,
                mu, nu, offA, offB, lda, ldb, mulA, mulB,
                rowA, rowB, &ia, &ib);
    }
}
```



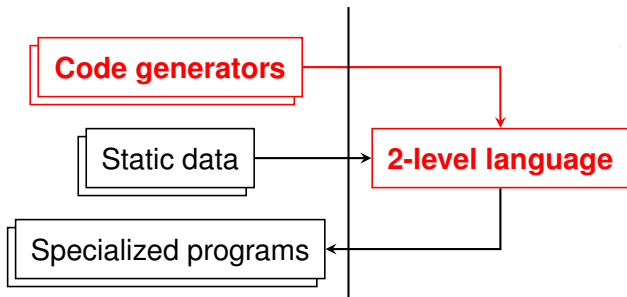
The need for domain-specific code generators



Want **safety**: generate well-formed programs only

Want **clarity**: generators resembles textbook algorithms

The need for domain-specific code generators



Want **safety**: generate well-formed programs only

Want **clarity**: generators resembles textbook algorithms

Our contribution: a two-level language

with a sound type system (for safety)

and delimited control operators (for clarity)

in which to express existing code-generation techniques

and domain-specific optimizations

Outline

▶ Example

Formalization

Gibonacci example

Like Fibonacci, but not always starting with 1 and 1.

```
let gib x y =  
  let rec loop n =  
    if n = 0 then x else  
    if n = 1 then y else  
    loop (n-1) + loop (n-2)  
  in loop
```

`gib 1 1 5` \longrightarrow 8

Other domains:

- ▶ Gaussian elimination
- ▶ Fast Fourier Transform
- ▶ Linear signal processing
- ▶ Embedded devices ...

Gibonacci example, specialized

Familiar from quasiquotation, macros, PE, or just printf.

```
let gib x y =  
  let rec loop n =  
    if n = 0 then x else  
    if n = 1 then y else  
    .<.~(loop (n-1)) + .~(loop (n-2))>.  
  in loop
```

```
.<fun x y -> .~(gib .<x>. .<y>. 5)>.
```

```
→ .<fun x_0 -> fun y_1 ->  
  (((y_1 + x_0) + y_1) + (y_1 + x_0)) +  
  ((y_1 + x_0) + y_1)>.
```

Code values can be open when evaluating under generated λ , but the generated code is always well-scoped, because names are generated and bound in one step.

Gibonacci example, memoized

Keep a memo table as mutable state.

```
let gib x y = let memo = new_memo () in
  let rec loop n =
    if n = 0 then x else
    if n = 1 then y else
    memo loop (n-1) + memo loop (n-2)
  in loop
```

`gib 1 1 5` \longrightarrow 8

Other domain-specific optimizations:

- ▶ Dynamic programming
- ▶ Pivoting matrices
- ▶ Simplifying arithmetic on complex roots of unity ...

Gibonacci example, specialized, memoized?

A naive combination duplicates code, as when unfolding in PE.

```
let gib x y = let memo = new_memo () in
  let rec loop n =
    if n = 0 then x else
    if n = 1 then y else
    .<~(memo loop (n-1)) + ~(memo loop (n-2))>.
  in loop
```

```
.<fun x y -> ~(gib .<x>. .<y>. 5)>.
```

```
→ .<fun x_0 -> fun y_1 ->
  (((y_1 + x_0) + y_1) + (y_1 + x_0)) +
  ((y_1 + x_0) + y_1)>.
```

Generating code fast is not generating fast code!

Two problems

1. Code in state voids safety, due to **scope extrusion**.

```
let r = ref .<1>. in
.<fun y -> .~(r := .<y>. ; .<()>.)>. ;
!r
```

→ .<y_1>.

2. Need to **insert let** at top, not to duplicate specialized code.

```
.<fun x y -> .~(gib .<x>. .<y>. 4)>.
```

```
→ .<fun x_0 -> fun y_1 ->
    let t_2 = y_1 + x_0 in
    let t_3 = t_2 + y_1 in t_3 + t_2>.
```

Two problems

1. Code in state voids safety, due to **scope extrusion**.

```
let r = ref .<1>. in
.<fun y -> .~(r := .<y>. ; .<()>.)>. ;
!r
→ .<y_1>.
```

2. Need to **insert let** at top, not to duplicate specialized code.

```
.<fun x y -> .~(gib .<x>. .<y>. 4)>.
→ .<fun x_0 -> fun y_1 ->
  let t_2 = y_1 + x_0 in
  let t_3 = t_2 + y_1 in t_3 + t_2>.
```

The diagram illustrates scope extrusion with two annotations: 'loop 2' and 'loop 3'. 'loop 2' has two red arrows pointing to the 'let t_2 = y_1 + x_0 in' line and the 't_2' variable in the expression 't_3 + t_2'. 'loop 3' has two red arrows pointing to the 'let t_3 = t_2 + y_1 in' line and the 't_3' variable in the expression 't_3 + t_2'. The 'let t_2 = y_1 + x_0 in' line is highlighted in yellow, and the 'let t_3 = t_2 + y_1 in t_3 + t_2' line is highlighted in orange.

(Similar: need to insert if/assert.)

Two solutions

1. Use CPS or monadic style to write the generator. (Match compiler, CPS translator (Danvy & Filinski), PE (Bondorf))

```
let gib x y =
  let rec loop n k =
    if n = 0 then k x else
    if n = 1 then k y else
    memo loop (n-1) (fun r1 ->
    memo loop (n-2) (fun r2 ->
    k .<~r1 + ~r2>.)
  in loop
```

Two solutions

1. Use CPS or monadic style to write the generator. (Match compiler, CPS translator (Danvy & Filinski), PE (Bondorf))

```
let gib x y =
  let rec loop n k =
    if n = 0 then k x else
    if n = 1 then k y else
    memo loop (n-1) (fun r1 ->
    memo loop (n-2) (fun r2 ->
    k .<~r1 + ~r2>.)
  in loop
```

```
loop 2 k table ≈ .<let t_2 = y_1 + x_0 in
  .~(k .<t_2>. table')>.
```

```
loop 3 k table' ≈ .<let t_3 = t_2 + y_1 in
  .~(k .<t_3>. table'')>.
```

Two solutions

1. Use CPS or monadic style to write the generator. (Match compiler, CPS translator (Danvy & Filinski), PE (Bondorf))

```
let gib x y =
  let rec loop n k =
    if n = 0 then k x else
    if n = 1 then k y else
    memo loop (n-1) (fun r1 ->
    memo loop (n-2) (fun r2 ->
    k .<~r1 + ~r2>.)
    in loop

.<fun x y -> .~(top_fn (gib .<x>. .<y>. 5))>.
→ .<fun x_0 -> fun y_1 ->
  let t_1 = y_1 in let t_0 = x_0 in
  let t_2 = t_1 + t_0 in
  let t_3 = t_2 + t_1 in
  let t_4 = t_3 + t_2 in t_4 + t_3>.
```

Two solutions

1. Use CPS or monadic style to write the generator. (Match compiler, CPS translator (Danvy & Filinski), PE (Bondorf))

```
let gib x y =
```

```
  let rec loop n k =
```

```
    if n = 0 then k x else
```

```
    if n = 1 then k y else
```

```
    memo loop (n-1) (fun r1 ->
```

```
    memo loop (n-2) (fun r2 ->
```

```
    k .<~r1 + ~r2>.)
```

```
  in loop
```

```
  .<fun x y -> .~(top_fn (gib .<x>. .<y>. 5))>.
```

```
  → .<fun x_0 -> fun y_1 ->
```

```
    let t_1 = y_1 in let t_0 = x_0 in
```

```
    let t_2 = t_1 + t_0 in
```

```
    let t_3 = t_2 + t_1 in
```

```
    let t_4 = t_3 + t_2 in t_4 + t_3>.
```

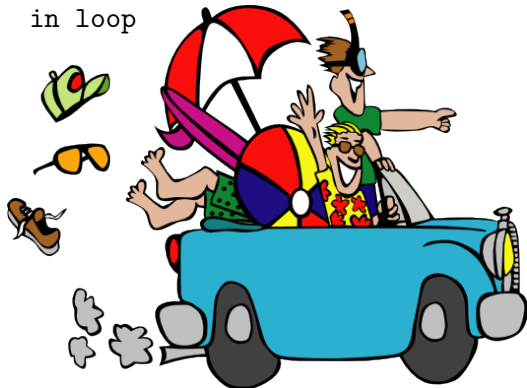


Two solutions

2. Use *delimited control operators* to hide CPS.

(CPS translator (Danvy & Filinski), PE (Lawall & Danvy))

```
let gib x y =  
  let rec loop n =  
    if n = 0 then x else  
    if n = 1 then y else  
    .<~(memo loop (n-1)) + ~(memo loop (n-2))>.  
  in loop
```



Two solutions

2. Use *delimited control operators* to hide CPS.

(CPS translator (Danvy & Filinski), PE (Lawall & Danvy))

```
let gib x y =  
  let rec loop n =  
    if n = 0 then x else  
    if n = 1 then y else  
    .<.(memo loop (n-1)) + .(memo loop (n-2))>.  
  in loop
```

$$\langle D[\text{loop } 2] \rangle \text{ table} \approx .\langle \text{let } t_2 = y_1 + x_0 \text{ in} \\ \sim(\langle D[.\langle t_2 \rangle.] \rangle \text{ table}') \rangle .$$
$$\langle D[\text{loop } 3] \rangle \text{ table}' \approx .\langle \text{let } t_3 = t_2 + y_1 \text{ in} \\ \sim(\langle D[.\langle t_3 \rangle.] \rangle \text{ table}'') \rangle .$$

Two solutions

2. Use *delimited control operators* to hide CPS.

(CPS translator (Danvy & Filinski), PE (Lawall & Danvy))

```
let gib x y =
  let rec loop n =
    if n = 0 then x else
    if n = 1 then y else
    .<~(memo loop (n-1)) + ~(memo loop (n-2))>.
  in loop

.<fun x y -> ~(top_fn (fun () -> gib .<x>. .<y>. 5))>.
→ .<fun x_0 -> fun y_1 ->
  let t_1 = y_1 in let t_0 = x_0 in
  let t_2 = t_1 + t_0 in
  let t_3 = t_2 + t_1 in
  let t_4 = t_3 + t_2 in t_4 + t_3>.
```

Two solutions

2. Use *delimited control operators* to hide CPS.

(CPS translator (Danvy & Filinski), PE (Lawall & Danvy))

```
let gib x y =  
  let rec loop n =  
    if n = 0 then x else  
    if n = 1 then y else  
    .<~(memo loop (n-1)) + ~(memo loop (n-2))>.  
  in loop
```

```
top_fn (fun () -> .<fun x y -> ~(gib .<x> . .<y> . 5)>.)
```

```
→ .<let t_1 = y_1 in let t_0 = x_0 in  
  let t_2 = t_1 + t_0 in  
  let t_3 = t_2 + t_1 in  
  let t_4 = t_3 + t_2 in  
  fun x_0 -> fun y_1 -> t_4 + t_3>.
```



Preventing scope extrusion



Custom generators

≠

Fixed generator



Our contribution:

For safety, simply **treat later binders as earlier delimiters** in the operational semantics and type system.

(Existing practice; Thiemann & Dussart's constraint on state)

Outline

Example

► **Formalization**

Our source language λ_1°

Expressions $e ::= x \mid i \mid e + e \mid \lambda x. e \mid \text{fix} \mid ee$
 $\mid (e, e) \mid \text{fst} \mid \text{snd} \mid \text{ifz } e \text{ then } e \text{ else } e$
 $\mid \text{出} \mid \{e\} \mid \langle e \rangle \mid \sim e$

$$C[(\lambda x. e) v] \rightsquigarrow C[e[x := v]] \quad (\beta_v)$$

\vdots

Our source language λ_1^\emptyset

Expressions $e ::= x \mid i \mid e + e \mid \lambda x. e \mid \text{fix} \mid ee$
| $(e, e) \mid \text{fst} \mid \text{snd} \mid \text{ifz } e \text{ then } e \text{ else } e$
| $\underbrace{\text{出} \mid \{e\}}_{\text{Delimited control}} \mid \underbrace{\langle e \rangle \mid \sim e}_{\text{Code generation}}$

(Felleisen, . . . , Danvy & Filinski) (Davies & Pfenning, . . . , Taha)

$$C[(\lambda x. e) v] \rightsquigarrow C[e[x := v]] \quad (\beta_v)$$

⋮

Staging

Two levels: present 0, future 1.

Expressions $e ::= x \mid i \mid e + e \mid \lambda x. e \mid \text{fix} \mid ee$
| $(e, e) \mid \text{fst} \mid \text{snd} \mid \text{ifz } e \text{ then } e \text{ else } e$
| $\underbrace{\text{⏏} \mid \{e\}}_{\text{Delimited control}} \mid \underbrace{\langle e \rangle \mid \sim e}_{\text{Code generation}}$

(Felleisen, . . . , Danvy & Filinski) (Davies & Pfenning, . . . , Taha)

$$C[(\lambda x. e) v] \rightsquigarrow C[e[x := v]] \quad (\beta_v)$$

⋮

Staging

Two levels: present 0, future 1.

Values $v^0 ::= x \mid \lambda x. e \mid \langle v^1 \rangle \mid \dots$
 $v^1 ::= x \mid \lambda x. v^1 \mid v^1 v^1 \mid \dots$

Contexts $C^0 ::= C^0[\square e] \mid C^0[v^0 \square] \mid C^1[\sim \square] \mid \square \mid \dots$
 $C^1 ::= C^1[\square e] \mid C^1[v^1 \square] \mid C^0[\langle \square \rangle] \mid \dots$

$$C^0[(\lambda x. e) v^0] \rightsquigarrow C^0[e[x := v^0]] \quad (\beta_v)$$

$$C^1[\sim \langle v^1 \rangle] \rightsquigarrow C^1[v^1] \quad (\sim)$$

\vdots

Staging

Two levels: present 0, future 1.

$$\begin{aligned} \text{let } f = \lambda x. x \text{ in } \langle \lambda t. \sim(f\langle t \rangle) \rangle &\rightsquigarrow_{\beta_v} \langle \lambda t. \sim((\lambda x. x)\langle t \rangle) \rangle \\ &\rightsquigarrow_{\beta_v} \langle \lambda t. \sim\langle t \rangle \rangle \\ &\rightsquigarrow_{\sim} \langle \lambda t. t \rangle \end{aligned}$$

$$\begin{aligned} C^0[(\lambda x. e) v^0] &\rightsquigarrow C^0[e[x := v^0]] && (\beta_v) \\ C^1[\sim\langle v^1 \rangle] &\rightsquigarrow C^1[v^1] && (\sim) \\ &\vdots && \end{aligned}$$

Control

Two operators: shift 出 , reset $\{ \}$.

Expressions $e ::= x \mid i \mid e + e \mid \lambda x. e \mid \text{fix} \mid ee$
 $\mid (e, e) \mid \text{fst} \mid \text{snd} \mid \text{ifz } e \text{ then } e \text{ else } e$
 $\mid \text{出} \mid \{e\} \mid \underbrace{\langle e \rangle}_{\text{Code generation}} \mid \underbrace{\sim e}_{\text{Code generation}}$

Delimited control **Code generation**

(Felleisen, ..., Danvy & Filinski) (Davies & Pfenning, ..., Taha)

$$C^0[(\lambda x. e) v^0] \rightsquigarrow C^0[e[x := v^0]] \quad (\beta_v)$$

$$C^1[\sim \langle v^1 \rangle] \rightsquigarrow C^1[v^1] \quad (\sim)$$

$$C^0[\{v^0\}] \rightsquigarrow C^0[v^0] \quad (\{\})$$

$$C^0[\{D[\text{出} v^0]\}] \rightsquigarrow C^0[\{v^0(\lambda x. \{D[x]\})\}] \quad (\text{出}^0)$$

\vdots

Control

Two operators: shift 出, reset { }.

$$\begin{aligned} \{1 + 1\} + 1 &\rightsquigarrow_+ \{2\} + 1 \\ &\rightsquigarrow_{\{\}} 2 + 1 \\ &\rightsquigarrow_+ 3 \end{aligned}$$

$$\begin{aligned} C^0[(\lambda x. e) v^0] &\rightsquigarrow C^0[e[x := v^0]] && (\beta_v) \\ C^1[\sim\langle v^1 \rangle] &\rightsquigarrow C^1[v^1] && (\sim) \\ C^0[\{v^0\}] &\rightsquigarrow C^0[v^0] && (\{\}) \\ C^0[\{D[\text{出} v^0]\}] &\rightsquigarrow C^0[\{v^0(\lambda x. \{D[x]\})\}] && (\text{出}^0) \\ &\vdots && \end{aligned}$$

Control

Two operators: shift 出 , reset $\{ \}$. Emulate state (Filinski).

$\text{const} = \lambda y. \lambda z. y$ $\text{get} = \text{出}(\lambda k. \lambda z. k z z)$ $\text{put} = \lambda z'. \text{出}(\lambda k. \lambda z. k z' z')$

$\{\text{const}(\text{get} + 40)\} 2 \rightsquigarrow_{\text{出}^0} \{(\lambda k. \lambda z. k z z)(\lambda x. \{\text{const}(x + 40)\})\} 2$
 $\rightsquigarrow_{\beta_v} \{\lambda z. (\lambda x. \{\text{const}(x + 40)\}) z z\} 2$
 $\rightsquigarrow_{\{ \}} (\lambda z. (\lambda x. \{\text{const}(x + 40)\}) z z) 2$
 $\rightsquigarrow_{\beta_v} (\lambda x. \{\text{const}(x + 40)\}) 2 2$
 $\rightsquigarrow_{\beta_v} \{\text{const}(2 + 40)\} 2 \rightsquigarrow_{\beta_v} \{\lambda z. 42\} 2 \rightsquigarrow^+ 42$

$$C^0[(\lambda x. e) v^0] \rightsquigarrow C^0[e[x := v^0]] \quad (\beta_v)$$

$$C^1[\sim\langle v^1 \rangle] \rightsquigarrow C^1[v^1] \quad (\sim)$$

$$C^0[\{v^0\}] \rightsquigarrow C^0[v^0] \quad (\{ \})$$

$$C^0[\{D[\text{出} v^0]\}] \rightsquigarrow C^0[\{v^0(\lambda x. \{D[x]\})\}] \quad (\text{出}^0)$$

\vdots

Control

Two operators: shift 出, reset { }. Emulate state (Filinski).

const = $\lambda y. \lambda z. y$ get = 出($\lambda k. \lambda z. k z z$) put = $\lambda z'. \text{出}(\lambda k. \lambda z. k z' z')$

$$\begin{aligned} \{\text{const}(\text{put}(\text{get} + 1) + \text{get})\} 2 &\rightsquigarrow^+ \{\text{const}(\text{put}(2 + 1) + \text{get})\} 2 \\ &\rightsquigarrow_+ \{\text{const}(\text{put } 3 + \text{get})\} 2 \\ &\rightsquigarrow^+ (\lambda x. \{\text{const}(x + \text{get})\}) 3 \ 3 \\ &\rightsquigarrow_{\beta_v} \{\text{const}(3 + \text{get})\} 3 \\ &\rightsquigarrow^+ \{\text{const}(3 + 3)\} 3 \rightsquigarrow^+ 6 \end{aligned}$$

$$\begin{aligned} C^0[(\lambda x. e) v^0] &\rightsquigarrow C^0[e[x := v^0]] && (\beta_v) \\ C^1[\sim\langle v^1 \rangle] &\rightsquigarrow C^1[v^1] && (\sim) \\ C^0[\{v^0\}] &\rightsquigarrow C^0[v^0] && (\{\}) \\ C^0[\{D[\text{出} v^0]\}] &\rightsquigarrow C^0[\{v^0(\lambda x. \{D[x]\})\}] && (\text{出}^0) \\ &\vdots && \end{aligned}$$

Staging + Control

Is scope extrusion possible?

$\{\text{const } (\text{let } x = \langle \lambda y. \sim(\text{put } \langle y \rangle) \rangle \text{ in get})\} \langle 0 \rangle$

$\rightsquigarrow^+ \{\text{const } (\text{let } x = \langle \lambda y. \sim(\langle y \rangle) \rangle \text{ in get})\} \langle y \rangle$

$\rightsquigarrow^+ \{\text{const get}\} \langle y \rangle$

$\rightsquigarrow^+ \{\text{const } \langle y \rangle\} \langle y \rangle$

$\rightsquigarrow^+ \langle y \rangle$

Staging + Control

Is scope extrusion possible? No. Level-1 λ delimits control.

$\{\text{const } (\text{let } x = \langle \lambda y. \sim(\text{put } \langle y \rangle) \rangle \text{ in get})\} \langle 0 \rangle$

$\rightsquigarrow^+ \{\text{const } (\text{let } x = \langle \lambda y. \sim\{(\lambda k. \lambda z. k \langle y \rangle \langle y \rangle)(\lambda x. \{\langle \sim x \rangle\})\} \rangle \text{ in get})\} \langle 0 \rangle$

Prevented in the operational semantics, the type system, and the CPS translation:

$$\begin{aligned} \llbracket \lambda x. e \rrbracket_1 &= \lambda k. k \langle \lambda x. \sim(\llbracket e \rrbracket_1(\lambda z. z)) \rangle \\ &\neq \lambda k. \llbracket e \rrbracket_1(\lambda z. k \langle \lambda x. \sim z \rangle) \end{aligned}$$

Applications

Can write:

- ▶ memoized Fibonacci generator
- ▶ a memoizing fixpoint combinator for code generation
- ▶ dynamic programming
- ▶ Gaussian elimination
- ▶ combinators for CPS translation and partial evaluation

Cannot write:

- ▶ loop-invariant code motion
- ▶ inserting `let/if/assert` at outermost possible scope

Type system

Environments contain bindings at two levels:

$$\Gamma ::= [] \mid \Gamma, x : \tau \mid \Gamma, \langle x : v \rangle$$

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash \lambda k. kx : (\tau \rightarrow \tau_0) \rightarrow \tau_0}$$

$$\frac{(\langle x : v \rangle) \in \Gamma}{\Gamma \vdash \lambda k. k\langle x \rangle : (\langle v \rangle \rightarrow \tau_0) \rightarrow \tau_0}$$

Type system

Environments contain bindings at two levels:

$$\Gamma ::= [] \mid \Gamma, x : \tau \mid \Gamma, \langle x : v \rangle$$

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash \llbracket x \rrbracket_0 : (\tau \rightarrow \tau_0) \rightarrow \tau_0}$$

$$\frac{(\langle x : v \rangle) \in \Gamma}{\Gamma \vdash \llbracket x \rrbracket_1 : (\langle v \rangle \rightarrow \tau_0) \rightarrow \tau_0}$$

Type system

Environments contain bindings at two levels:

$$\Gamma ::= [] \mid \Gamma, x : \tau \mid \Gamma, \langle x : v \rangle$$

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau ; \tau_0} \qquad \frac{(\langle x : v \rangle) \in \Gamma}{\Gamma \vdash x : v ; \tau_0 ;}$$

Type system

Environments contain bindings at two levels:

$$\Gamma ::= [] \mid \Gamma, x : \tau \mid \Gamma, \langle x : v \rangle$$

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau ; \tau_0} \quad \frac{(\langle x : v \rangle) \in \Gamma}{\Gamma \vdash x : v ; \tau_0 ;}$$
$$\frac{\Gamma, x : \tau \vdash \llbracket e \rrbracket_0 : (\tau' \rightarrow \tau_1) \rightarrow \tau_1}{\Gamma \vdash \lambda k. k(\lambda x. \llbracket e \rrbracket_0) : ((\tau \rightarrow (\tau' \rightarrow \tau_1) \rightarrow \tau_1) \rightarrow \tau_0) \rightarrow \tau_0}$$

Type system

Environments contain bindings at two levels:

$$\Gamma ::= [] \mid \Gamma, x : \tau \mid \Gamma, \langle x : v \rangle$$

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau ; \tau_0} \quad \frac{(\langle x : v \rangle) \in \Gamma}{\Gamma \vdash x : v ; \tau_0 ;}$$
$$\frac{\Gamma, x : \tau \vdash \llbracket e \rrbracket_0 : (\tau' \rightarrow \tau_1) \rightarrow \tau_1}{\Gamma \vdash \llbracket \lambda x. e \rrbracket_0 : ((\tau \rightarrow (\tau' \rightarrow \tau_1) \rightarrow \tau_1) \rightarrow \tau_0) \rightarrow \tau_0}$$

Type system

Environments contain bindings at two levels:

$$\Gamma ::= [] \mid \Gamma, x : \tau \mid \Gamma, \langle x : v \rangle$$

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau ; \tau_0} \quad \frac{\langle (x : v) \rangle \in \Gamma}{\Gamma \vdash x : v ; \tau_0 ;}$$
$$\frac{\Gamma, x : \tau \vdash e : \tau' ; \tau_1}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau' / \tau_1 ; \tau_0}$$

Type system

Environments contain bindings at two levels:

$$\Gamma ::= [] \mid \Gamma, x : \tau \mid \Gamma, \langle x : v \rangle$$

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau ; \tau_0} \quad \frac{\langle \langle x : v \rangle \rangle \in \Gamma}{\Gamma \vdash x : v ; \tau_0 ;}$$
$$\frac{\Gamma, x : \tau \vdash e : \tau' ; \tau_1}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau' / \tau_1 ; \tau_0}$$
$$\frac{\Gamma, \langle x : v \rangle \vdash \llbracket e \rrbracket_1 : (\langle v' \rangle \rightarrow \langle v' \rangle) \rightarrow \langle v' \rangle}{\Gamma \vdash \lambda k. k \langle \lambda x. \sim(\llbracket e \rrbracket_1(\lambda z. z)) \rangle : (\langle v \rightarrow v' \rangle \rightarrow \tau_0) \rightarrow \tau_0}$$

Type system

Environments contain bindings at two levels:

$$\Gamma ::= [] \mid \Gamma, x : \tau \mid \Gamma, \langle x : v \rangle$$

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau ; \tau_0} \qquad \frac{\langle \langle x : v \rangle \rangle \in \Gamma}{\Gamma \vdash x : v ; \tau_0 ;}$$
$$\frac{\Gamma, x : \tau \vdash e : \tau' ; \tau_1}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau' / \tau_1 ; \tau_0}$$
$$\frac{\Gamma, \langle x : v \rangle \vdash \llbracket e \rrbracket_1 : (\langle v' \rangle \rightarrow \langle v' \rangle) \rightarrow \langle v' \rangle}{\Gamma \vdash \llbracket \lambda x. e \rrbracket_1 : (\langle v \rightarrow v' \rangle \rightarrow \tau_0) \rightarrow \tau_0}$$

Type system

Environments contain bindings at two levels:

$$\Gamma ::= [] \mid \Gamma, x : \tau \mid \Gamma, \langle x : v \rangle$$

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau ; \tau_0} \qquad \frac{\langle (x : v) \rangle \in \Gamma}{\Gamma \vdash x : v ; \tau_0 ;}$$
$$\frac{\Gamma, x : \tau \vdash e : \tau' ; \tau_1}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau' / \tau_1 ; \tau_0}$$
$$\frac{\Gamma, \langle x : v \rangle \vdash e : v' ; \langle v' \rangle ;}{\Gamma \vdash \lambda x. e : v \rightarrow v' ; \tau_0 ;}$$

Conclusion



The first language for code generators with

- ▶ **a sound type system, for safety:**
generate well-formed programs only
- ▶ **delimited control operators, for clarity:**
generators resemble textbook algorithms

Key: **treat later binders as earlier delimiters**

Implemented in MetaOCaml (manual treatment)
and in Twelf



Extensions wanted:

- ▶ control effects beyond nearest delimiter
- ▶ changing answer types
- ▶ (answer-type) polymorphism
- ▶ more than 2 levels
- ▶ eval and polymorphic lift