Shifting the stage

Staging with delimited control

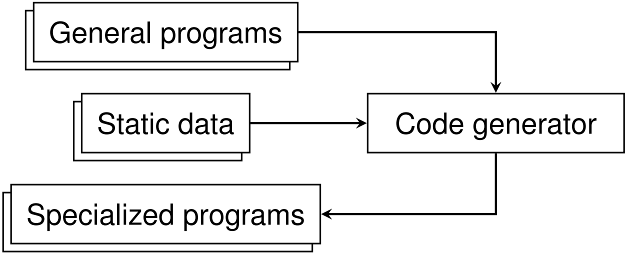
Yukiyoshi Kameyama
University of Tsukuba
`kameyama@acm.org`

Oleg Kiselyov
FNMOC
`oleg@pobox.com`

Chung-chieh Shan
Rutgers
`ccshan@rutgers.edu`

PEPM, 20 January 2009

The need for domain-specific code generators



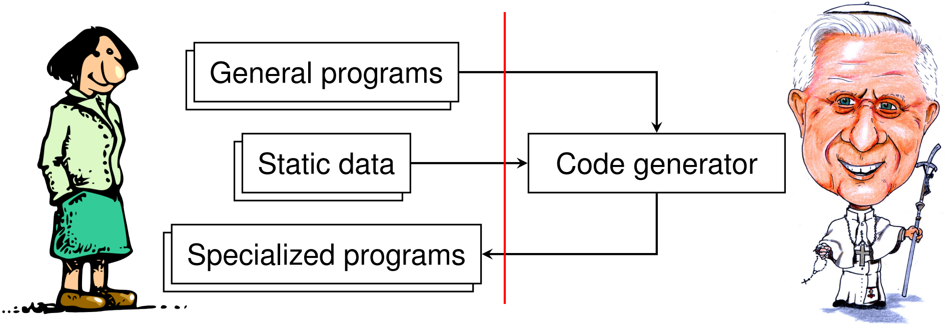
General programs

Static data

Code generator

Specialized programs

The need for domain-specific code generators



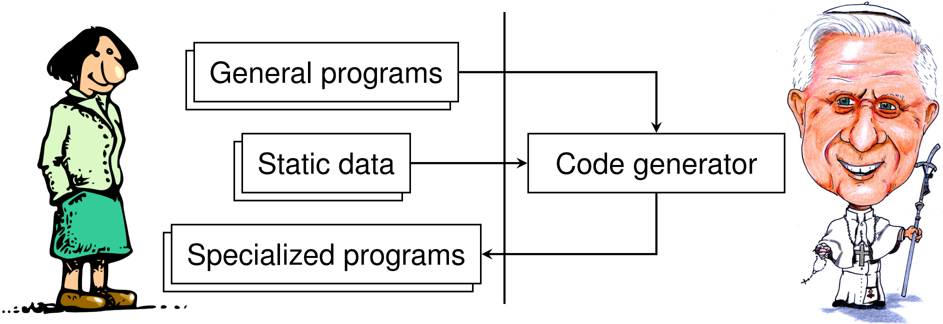
General programs

Static data

Code generator

Specialized programs

The need for domain-specific code generators



General programs

Static data

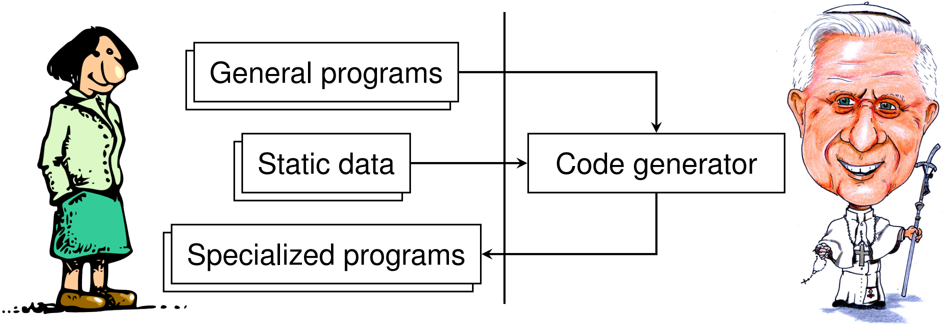
Code generator

Specialized programs

Optimizations specific to . . .

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

The need for domain-specific code generators



General programs

Static data

Code generator

Specialized programs

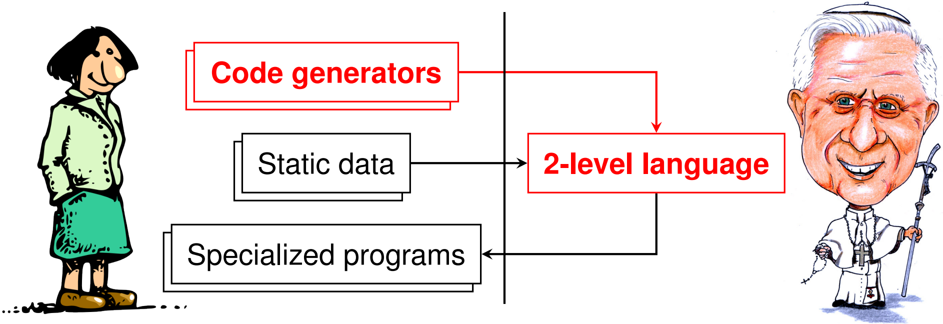
Optimizations specific to . . .

- 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



Code generators

Static data

2-level language

Specialized programs

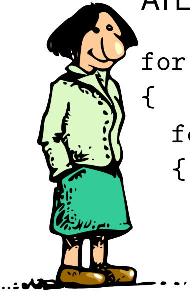
Optimizations specific to . . .

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

Generate code using . . .

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

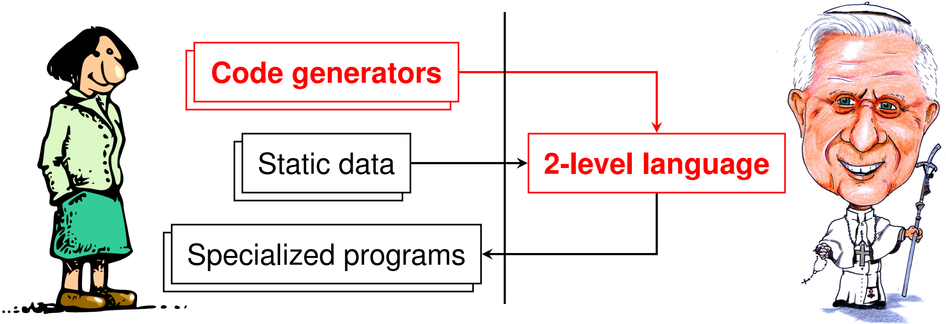
The need for domain-specific code generators



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



Code generators

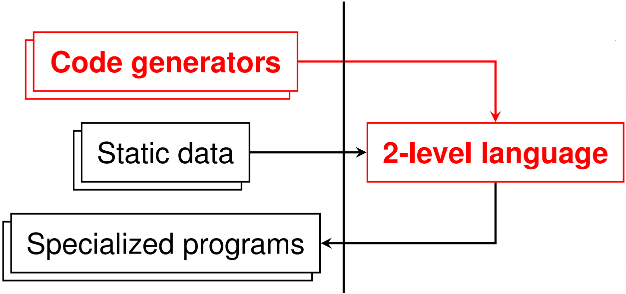
Static data

2-level language

Specialized programs

Want **safety**: generate well-formed programs only
Want **clarity**: generators resembles textbook algorithms

The need for domain-specific code generators



Code generators

Static data

2-level language

Specialized programs

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` ⟶ `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` ⟶ `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>.
```

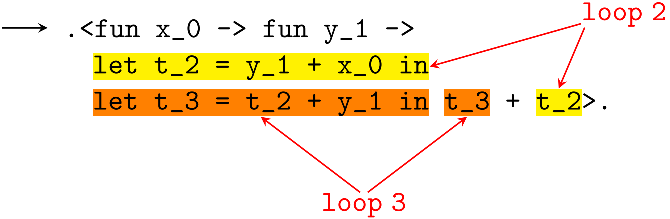
Two problems

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>.
```

(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))

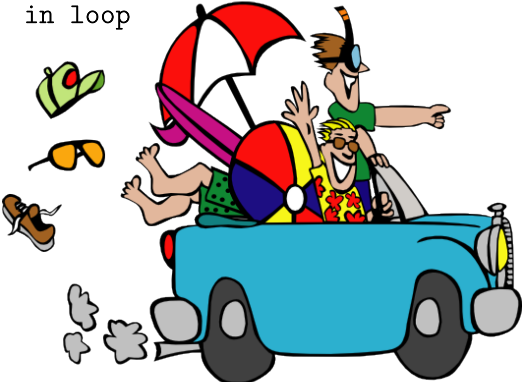
```
let gib x y =
  let rec loop n k =
    if n = 0 then k x else
    if n = 1 then k y else
    memo loop (n-1) (fun r1 ->
    memo loop (n-2) (fun r2 ->
    k .<.~r1 + .~r2>.))
  in loop
```

```
.<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
```

```
⟨D[loop 2]⟩ table  ≈ .<let t_2 = y_1 + x_0 in
                        .~(⟨D[.<t_2>.]⟩ table')>.
⟨D[loop 3]⟩ table' ≈ .<let t_3 = t_2 + y_1 in
                        .~(⟨D[.<t_3>.]⟩ table'')>.
```

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>.
```

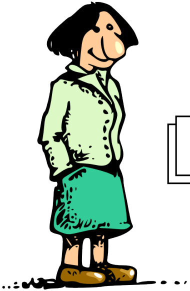
Two solutions

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 $\neq$ 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 $\lambda_1^{\varnothing}$

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

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

Our source language $\lambda_1^{\varnothing}$

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

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

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

$$\vdots$$

Staging

Two levels: present 0, future 1.

$$
\begin{aligned}
\text{Expressions}\quad e \; ::= \;& x \mid i \mid e + e \mid \lambda x.\, e \mid \mathsf{fix} \mid e e \\
\mid\;& (e, e) \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{ifz}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
\mid\;& \underbrace{出 \mid \{e\}}_{\text{Delimited control}} \mid \underbrace{\langle e \rangle \mid \sim e}_{\text{Code generation}}
\end{aligned}
$$

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

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

$$\vdots$$

Staging

Two levels: present 0, future 1.

$$\begin{array}{llcl} \text{Values} & v^0 & ::= & x \mid \lambda x.\, e \mid \langle v^1 \rangle \mid \cdots \\ & v^1 & ::= & x \mid \lambda x.\, v^1 \mid v^1 v^1 \mid \cdots \end{array}$$

$$\begin{array}{llcl} \text{Contexts} & C^0 & ::= & C^0[\square e] \mid C^0[v^0 \square] \mid C^1[{\sim}\square] \mid \square \mid \cdots \\ & C^1 & ::= & C^1[\square e] \mid C^1[v^1 \square] \mid C^0[\langle \square \rangle] \mid \cdots \end{array}$$

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

$$C^1[{\sim}\langle v^1 \rangle] \rightsquigarrow C^1[v^1] \qquad ({\sim})$$

$$\vdots$$

Staging

Two levels: present 0, future 1.

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

$$C^0\left[(\lambda x.\, e)\, v^0\right] \rightsquigarrow C^0\left[e[x := v^0]\right] \qquad (\beta_v)$$

$$C^1\left[{\sim}\langle v^1 \rangle\right] \rightsquigarrow C^1\left[v^1\right] \qquad (\sim)$$

$$\vdots$$

Control

Two operators: shift 出, reset { }.

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

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

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

Control

Two operators: shift 出, reset { }.

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

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

Control

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

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

$$\begin{aligned}
\{\text{const}\,(\text{get} + 40)\}\,2 &\rightsquigarrow_{\text{出}^0} \{(\lambda k.\,\lambda z.\,kzz)(\lambda x.\,\{\text{const}(x + 40)\})\}\,2\\
&\rightsquigarrow_{\beta_v} \{\lambda z.\,(\lambda x.\,\{\text{const}(x + 40)\})zz\}\,2\\
&\rightsquigarrow_{\{\}} (\lambda z.\,(\lambda x.\,\{\text{const}(x + 40)\})zz)\,2\\
&\rightsquigarrow_{\beta_v} (\lambda x.\,\{\text{const}(x + 40)\})\,2\,2\\
&\rightsquigarrow_{\beta_v} \{\text{const}(2 + 40)\}\,2 \rightsquigarrow_{\beta_v} \{\lambda z.\,42\}\,2 \rightsquigarrow^{+} 42
\end{aligned}$$

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

Control

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

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

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

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

Staging + Control

Is scope extrusion possible?

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

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

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

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

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

Staging + Control

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

$$\begin{aligned} &\{\mathsf{const}\ (\mathsf{let}\ x = \langle \lambda y.\, {\sim}(\mathsf{put}\,\langle y\rangle)\rangle\ \mathsf{in}\ \mathsf{get})\}\ \langle 0\rangle \\ \rightsquigarrow^{+}\ &\{\mathsf{const}\ (\mathsf{let}\ x = \langle \lambda y.\, {\sim}\{(\lambda k.\, \lambda z.\, k\langle y\rangle\langle y\rangle)(\lambda x.\, \{\langle {\sim}x\rangle\})\}\rangle\ \mathsf{in}\ \mathsf{get})\}\ \langle 0\rangle \end{aligned}$$

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

$$\begin{aligned} [\![\lambda x.\, e]\!]_1 &= \lambda k.\, k\,\langle \lambda x.\, {\sim}([\![e]\!]_1(\lambda z.\, z))\rangle \\ &\neq \lambda k.\, [\![e]\!]_1(\lambda z.\, k\,\langle \lambda x.\, {\sim}z\rangle) \end{aligned}$$

Applications

Can write:

- memoized Gibonnacci 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:

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

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash \lambda k.\, kx : (\tau \to \tau_0) \to \tau_0} \qquad \frac{(\langle x : \upsilon \rangle) \in \Gamma}{\Gamma \vdash \lambda k.\, k\langle x \rangle : (\langle \upsilon \rangle \to \tau_0) \to \tau_0}$$

Type system

Environments contain bindings at two levels:

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

Pull back from CPS translation.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash \ [\![x]\!]_0 \ : (\tau \to \tau_0) \to \tau_0} \qquad \frac{(\langle x : \upsilon \rangle) \in \Gamma}{\Gamma \vdash \ [\![x]\!]_1 \ : (\langle \upsilon \rangle \to \tau_0) \to \tau_0}$$

Type system

Environments contain bindings at two levels:

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

Pull back from CPS translation.

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

Type system

Environments contain bindings at two levels:

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

Pull back from CPS translation.

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

$$\frac{\Gamma, x : \tau \vdash [\![e]\!]_0 : (\tau' \to \tau_1) \to \tau_1}{\Gamma \vdash \lambda k.\, k(\lambda x.\, [\![e]\!]_0) : ((\tau \to (\tau' \to \tau_1) \to \tau_1) \to \tau_0) \to \tau_0}$$

Type system

Environments contain bindings at two levels:

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

Pull back from CPS translation.

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

$$\frac{\Gamma, x : \tau \vdash [\![e]\!]_0 : (\tau' \to \tau_1) \to \tau_1}{\Gamma \vdash \quad [\![\lambda x.\, e]\!]_0 \quad : ((\tau \to (\tau' \to \tau_1) \to \tau_1) \to \tau_0) \to \tau_0}$$

Type system

Environments contain bindings at two levels:

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

Pull back from CPS translation.

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

$$\frac{\Gamma, x : \tau \vdash \quad e \quad : \quad \tau' \,;\, \tau_1}{\Gamma \vdash \quad \lambda x.\, e \quad : \quad \tau \to \tau'/\tau_1 \,;\, \tau_0}$$

Type system

Environments contain bindings at two levels:

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

Pull back from CPS translation.

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

$$\frac{\Gamma, x : \tau \vdash \quad e \quad : \quad \tau' \ ; \tau_1}{\Gamma \vdash \quad \lambda x.\, e \quad : \quad \tau \to \tau' / \tau_1 \ ; \tau_0}$$

$$\frac{\Gamma, \langle x : \upsilon \rangle \vdash [\![e]\!]_1 : (\langle \upsilon' \rangle \to \langle \upsilon' \rangle) \to \langle \upsilon' \rangle}{\Gamma \vdash \lambda k.\, k \, \langle \lambda x.\, {\sim}([\![e]\!]_1(\lambda z.\, z)) \rangle : (\langle \upsilon \to \upsilon' \rangle \to \tau_0) \to \tau_0}$$

Type system

Environments contain bindings at two levels:

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

Pull back from CPS translation.

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

$$\frac{\Gamma, x : \tau \vdash \quad e \quad : \quad \tau' \ ; \ \tau_1}{\Gamma \vdash \quad \lambda x.\, e \quad : \quad \tau \to \tau' / \tau_1 \ ; \ \tau_0}$$

$$\frac{\Gamma, \langle x : \upsilon \rangle \vdash [\![e]\!]_1 : (\langle \upsilon' \rangle \to \langle \upsilon' \rangle) \to \langle \upsilon' \rangle}{\Gamma \vdash \quad [\![\lambda x.\, e]\!]_1 \quad : (\langle \upsilon \to \upsilon' \rangle \to \tau_0) \to \tau_0}$$

Type system

Environments contain bindings at two levels:

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

Pull back from CPS translation.

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

$$\frac{\Gamma, x : \tau \vdash \ e \ : \ \tau' \ ; \ \tau_1}{\Gamma \vdash \ \lambda x.\, e \ : \ \tau \to \tau' / \tau_1 \ ; \ \tau_0}$$

$$\frac{\Gamma, \langle x : \upsilon \rangle \vdash \ e \ : \ \upsilon' \ ; \ \langle \upsilon' \rangle \ ;}{\Gamma \vdash \ \lambda x.\, e \ : \ \upsilon \to \upsilon' \ ; \ \tau_0 \ ;}$$

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