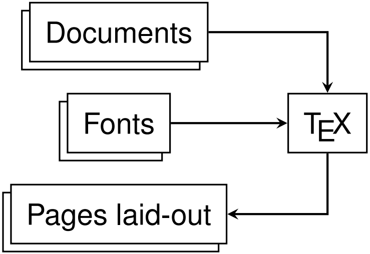
Typed metaprogramming with effects

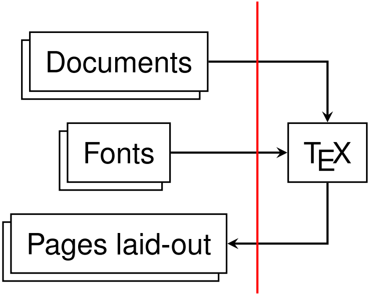
Chung-chieh Shan (Rutgers University)
with Chris Barker, Yukiyoshi Kameyama, Oleg Kiselyov

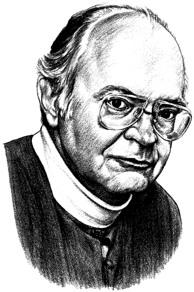
LFMTP
14 July 2010

Accidental language design

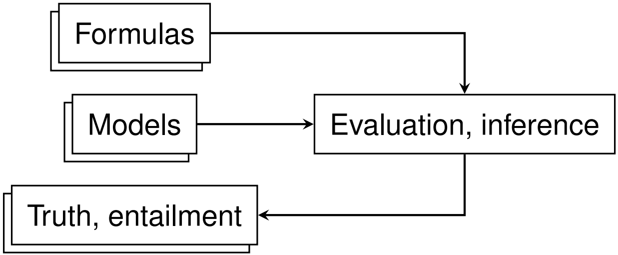


Accidental language design

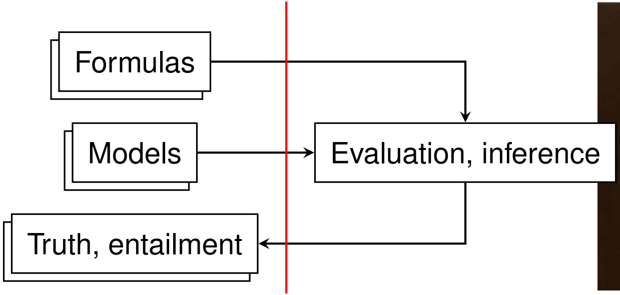




Accidental language design



Accidental language design

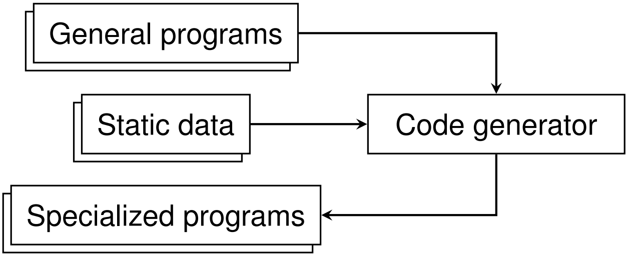


"It is probably more perspicuous to proceed indirectly, by

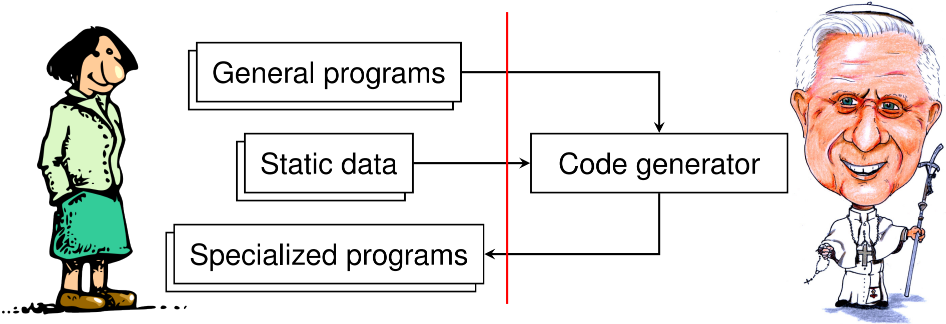
1. setting up a certain simple artificial language, that of tensed intensional logic,
2. giving the semantics of that language, and
3. interpreting English indirectly by showing in a rigorous way how to translate it into the artificial language.

This is the procedure we shall adopt . . . " —*Richard Montague*

Accidental language design



Accidental language design



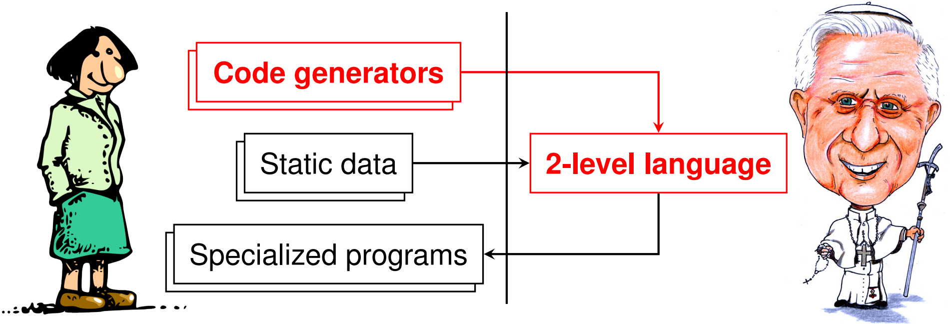
Optimizations specific to . . .

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

Generate code using . . .

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

Accidental language design



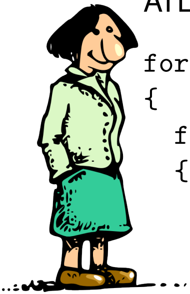
Optimizations specific to . . .

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

Generate code using . . .

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

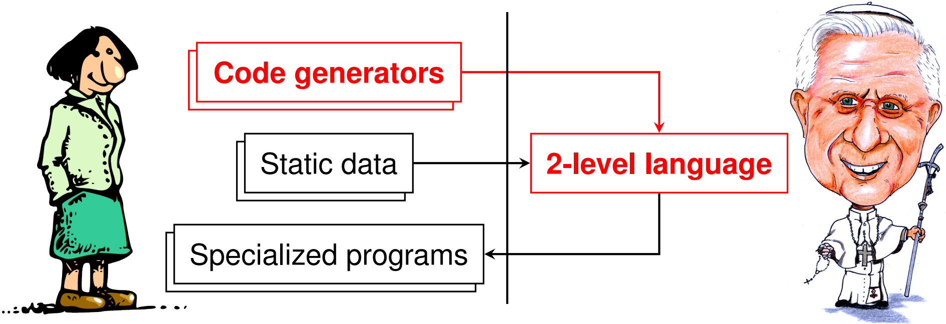
Accidental language design



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);
  }
}
```

Accidental language design



Want **safety**: generate well-formed programs only
$\leftarrow$ track object variable bindings

Want **clarity**: generators resemble textbook algorithms
$\leftarrow$ provide delimited control operators

Outline

- **Delimited control for program generation**
  - Example
  - Formalization
- Natural-language semantics
  - Delimited control
  - Quotation
  - Variable binding
- Breaking the fourth wall
  - Contextual modalities
  - Environment classifiers

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.
Binding context follows evaluation context, implicitly!

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>.
```

loop 2 (→ `let t_2 = y_1 + x_0 in`, `t_2`)

loop 3 (→ `let t_3 = t_2 + y_1 in`, `t_3`)

(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*″`)>.`

Importing `k` under `let` is ok because code is opaque!

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))

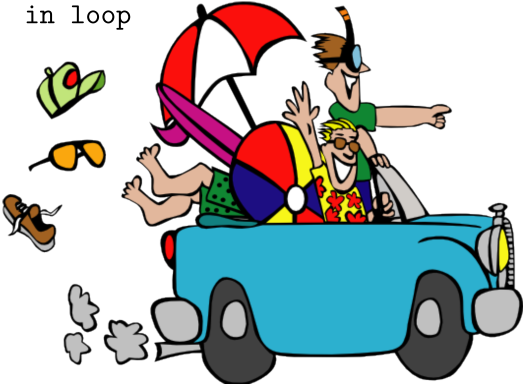
```
let gib x y =
  let rec loop n k =
    if n = 0 then k x else
    if n = 1 then k y else
    memo loop (n-1) (fun r1 ->
    memo loop (n-2) (fun r2 ->
    k .<.~r1 + .~r2>.))
  in loop
```

```
.<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
```

$\langle D[$`loop 2`$]\rangle$ *table* $\approx$ `.<let t_2 = y_1 + x_0 in`
`.~(`$\langle D[$`.<t_2>.`$]\rangle$ *table*$'$`)>.`

$\langle D[$`loop 3`$]\rangle$ *table*$'$ $\approx$ `.<let t_3 = t_2 + y_1 in`
`.~(`$\langle D[$`.<t_3>.`$]\rangle$ *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

Low-hanging fruit:

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

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

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

Expressions $e ::= x \mid i \mid e + e \mid \lambda x.\, e \mid \mathsf{fix} \mid e e$
$\mid (e, e) \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{ifz}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$
$\mid$ 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 $\lambda_1^{\varnothing}$

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

Delimited control (Felleisen, . . . , Danvy & Filinski)

Code generation (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.

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

Delimited control (Felleisen, . . . , Danvy & Filinski)

Code generation (Davies & Pfenning, . . . , Taha)

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

$$\vdots$$

Staging

Two levels: present 0, future 1.

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

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

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

Staging

Two levels: present 0, future 1.

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

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

Control

Two operators: shift 出, reset { }.

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

Delimited control (Felleisen, . . . , Danvy & Filinski)

Code generation (Davies & Pfenning, . . . , Taha)

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

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).

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

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

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

Control

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

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

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

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

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.

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

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

Staging + Control

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

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

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

Can write:

- memoizing fixpoint, CPS translation, partial evaluation
- dynamic programming
- Gaussian elimination
- Markov models with ‘symbolic’ matrix multiplications

Cannot write:

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

$$\{\langle \lambda i.\, {\sim}(\text{出} \lambda k.\, \langle \text{let}\ x = 40 + 2\ \text{in}\ {\sim}(k\langle i + x \rangle) \rangle) \rangle\}$$

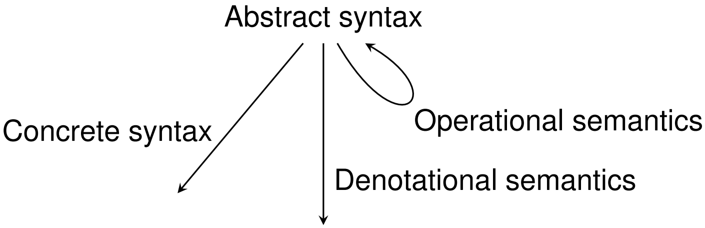
Outline

Formal linguistics

Goal: relate forms to meanings in a concise specification.
Science, rather than engineering.

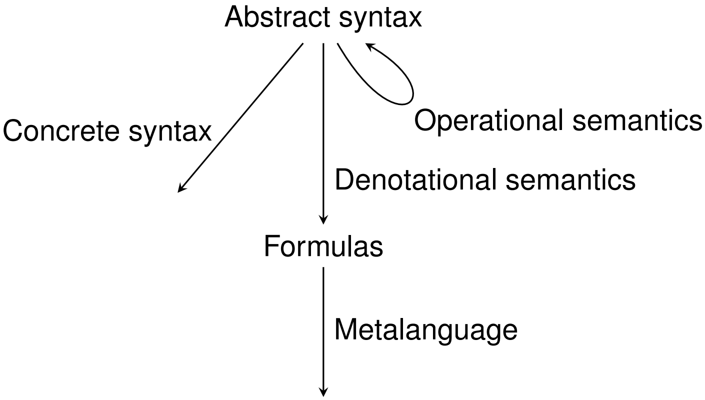
Formal linguistics

Goal: relate forms to meanings in a concise specification.
Science, rather than engineering.



Formal linguistics

Goal: relate forms to meanings in a concise specification.
Science, rather than engineering.



Cf. introductory logic.

Truth and entailment

Alice left and Bob left.

$\downarrow$

$left\ a \wedge left\ b$

Truth and entailment

$$
\begin{array}{ccc}
\text{Alice left and Bob left.} & \longmapsfrom & \text{Alice left.} \\
\downarrow & & \downarrow \\
\mathit{left}\ a \wedge \mathit{left}\ b & \longmapsfrom & \mathit{left}\ a
\end{array}
$$

Truth and entailment

Alice left and Bob left. ⟻ Alice left.

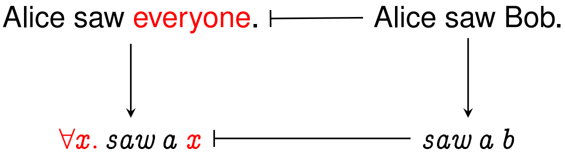
↓ ↓

$\mathit{left}\ a \wedge \mathit{left}\ b \longmapsfrom \mathit{left}\ a$

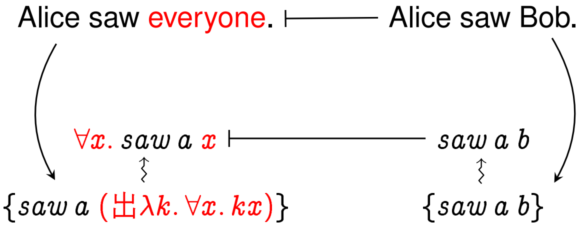
Alice and Bob left. ⟻ Alice left.

Alice and Bob met. ⟻̸ Alice met.

Delimited control for quantifiers



Delimited control for quantifiers



Delimited control for quantifiers

Alice saw everyone. ⟼ Alice saw Bob.

$\forall x.\, saw\; a\; x \longmapsfrom saw\; a\; b$

$\{saw\; a\; (出 \lambda k.\, \forall x.\, kx)\}$ $\qquad$ $\{saw\; a\; b\}$

Everyone saw someone.

$\forall x.\, \exists y.\, saw\; x\; y$

$\{saw\; (出 \lambda k.\, \forall x.\, kx)\; (出 \lambda k.\, \exists y.\, ky)\}$

Delimited control for quantifiers

Alice saw everyone. $\longmapsfrom$ Alice saw Bob.

$$\forall x.\ saw\ a\ x \longmapsfrom saw\ a\ b$$

$$\{saw\ a\ (出\lambda k.\ \forall x.\ kx)\} \rightsquigarrow \forall x.\ saw\ a\ x \qquad \{saw\ a\ b\} \rightsquigarrow saw\ a\ b$$

Everyone saw someone.

$$\forall x.\ \exists y.\ saw\ x\ y$$

$$\{saw\ (出\lambda k.\ \forall x.\ kx)\ (出\lambda k.\ \exists y.\ ky)\} \rightsquigarrow \forall x.\ \exists y.\ saw\ x\ y$$

Simulate other **linguistic side effects**: pronouns, questions, . . .

Evaluation order

Surface scope is preferred over **inverse scope**:

- Everyone saw someone.

Anaphora is preferred over **cataphora**:

- Everyone's father saw her mother.
  $*$ Her father saw everyone's mother.

Gap tends to precede **wh-phrase**:

- Who do you think saw what?
  $*$ What do you think who saw?

Reuse the same default of left-to-right evaluation for a more concise explanation.

Outline

Varieties of quotation

'Bachelor' has eight letters.

$\downarrow$ pure

$\textit{has-8-letters}$ 'bachelor'

Quine says 'quotation has a certain anomalous feature'.

$\downarrow$ direct

$\textit{say } q$ ⟨quotation has a certain anomalous feature⟩

Varieties of quotation

‘Bachelor’ has eight letters.

$\downarrow$ pure

$\textit{has-8-letters}$ ‘bachelor’

Quine says ‘quotation has a certain anomalous feature’.

$\downarrow$ direct

$\textit{say}\ q\ \langle\texttt{quotation has a certain anomalous feature}\rangle$

Quine says quotation has a certain anomalous feature.

$\downarrow$ indirect

$\textit{say}\ q\ (\textit{has-a-certain-anomalous-feature quotation})$

Varieties of quotation

'Bachelor' has eight letters.

$\downarrow$ pure

$\textit{has-8-letters}$ 'bachelor'

Quine says 'quotation has a certain anomalous feature'.

$\downarrow$ direct

$\textit{say}\ q$ ⟨quotation has a certain anomalous feature⟩

Quine says quotation has a certain anomalous feature.

$\downarrow$ indirect

$\textit{say}\ q\ (\textit{has-a-certain-anomalous-feature quotation})$

Quine says quotation 'has a certain anomalous feature'.

$\downarrow$ mixed

$\textit{say}\ q$ (⟨has a certain anomalous feature⟩ $\textit{quotation}$) ???

Mixing mention and use

Quine says quotation ‘has a certain anomalous feature’.

↓ mixed

$\ldots$ (eval q ⟨has a certain anomalous feature⟩) $\ldots$

Bush is proud of his ‘eckullectic’ reading list.

↓ mixed

$\ldots$ (eval b ⟨eckullectic⟩) $\ldots$

Mixing mention and use

Quine says quotation ‘has a certain anomalous feature’.

↓ mixed

…(eval q ⟨has a certain anomalous feature⟩)…

Bush is proud of his ‘eckullectic’ reading list.

↓ mixed

…(eval b ⟨eckullectic⟩)…

Yet Cheney’s reading list is far more ‘eckullectic’, not to mention longer.

↓ mixed

…(eval b ⟨eckullectic⟩)…

Program generation in natural language

Bush boasted of ‘my [Cheney’s favorite adjective] reading list’.

$\downarrow$ syntactic unquotation

$$\ldots \sim(\mathit{favorite\ adjective}\ c) \ldots$$

Bush boasted of ‘my [eclectic] reading list’.

$\downarrow$ semantic unquotation

$$\ldots \sim(出\lambda k.\, \exists x.\, \mathtt{eval}\ b\ x = \mathit{eclectic} \wedge k x) \ldots$$

Program generation in natural language

Bush boasted of ‘my [Cheney’s favorite adjective] reading list’.

↓ syntactic unquotation

$$\ldots \sim(\mathit{favorite\ adjective\ c}) \ldots$$

Bush boasted of ‘my [eclectic] reading list’.

↓ semantic unquotation

$$\ldots \sim(\text{出}\lambda k.\,\exists x.\,\mathsf{eval}\ b\ x = \mathit{eclectic} \wedge kx) \ldots$$

Bush complained about the ‘utterly [inaudible] loudspeakers’ in the room.

$$\ldots \sim(\text{出}\lambda k.\,\exists x.\,\mathit{inaudible}\ x \wedge kx) \ldots$$

$$\ldots \sim(\text{出}\lambda k.\,\exists x.\,\mathsf{eval}\ b\ x = \mathit{inaudible} \wedge kx) \ldots$$

Variable binding in natural language

The teacher praised every boy who did his homework.

$$\downarrow$$

$$\ldots (出 \lambda k.\, \forall x.\, (\mathit{boy}\ x \wedge \mathit{did}\ x\ (\mathit{homework}\ x)) \to k x) \ldots$$

The teacher praised ‘every boy who did [his homework]’.

$$\downarrow$$

$$\ldots \mathtt{eval}\ t\ \langle 出 \lambda k.\, \forall x.\, (\mathit{boy}\ x \wedge \mathit{did}\ x \sim\!\ldots) \to k x\rangle \ldots$$

Inverse quantifier scope

It is an attractive scientific hypothesis that evaluation order is *always* from left to right.

Everyone saw someone.

$$\{saw\,(\text{出}\lambda k.\,\forall x.\,kx)\,(\text{出}\lambda k.\,\exists y.\,ky)\} \rightsquigarrow \forall x.\,\exists y.\,saw\;x\;y$$

$$\{\langle\{saw\,(\text{出}\lambda k.\,\forall x.\,kx)\,{\sim}(\text{出}\lambda k.\,\langle\exists y.\,{\sim}(ky)\rangle)\}\rangle\}$$

$$\rightsquigarrow \langle\exists y.\,saw\,(\text{出}\lambda k.\,\forall x.\,kx)\;y\rangle$$

However, reducing generated shift statically is hard.

$$\{\langle\forall x.\,saw\;x\,{\sim}(\text{出}\lambda k.\,\langle\exists y.\,{\sim}(ky)\rangle)\rangle\}$$

If only . . .

Outline

Delimited control for program generation

- Example
- Formalization

Natural-language semantics

- Delimited control
- Quotation
- Variable binding

▶ **Breaking the fourth wall**

- Contextual modalities
- Environment classifiers

Loop-invariant code motion

We want:

$$\langle \lambda i.\, \text{let } y = i + (\ldots 40 + 2 \ldots) \text{ in } y + y \rangle \rightsquigarrow$$
$$\langle \text{let } x = 40 + 2 \text{ in } \lambda i.\, \text{let } y = i + x \text{ in } y + y \rangle$$

Or in a two-level calculus:

$$\underline{\lambda} i.\, (\underline{\lambda} y.\, y + y)(i + (\ldots \underline{40} + \underline{2} \ldots)) \rightsquigarrow$$

Or in CPS:

$$\begin{aligned} &(\underline{\lambda} i.\, \lambda k.\, (\underline{\lambda} y.\, \lambda k'.\, k'(y + y))(\lambda l. \\ &\qquad (\ldots \underline{40} + \underline{2} \ldots)(\lambda x. \\ &\qquad k(l(i + x))))) \\ &\quad (\lambda z.\, z) \rightsquigarrow \end{aligned}$$

Contextual modalities

In a two-level calculus:

$$\underline{\lambda} i.\, (\underline{\lambda} y.\, y + y) \\ (i + (\ldots \underline{40} + \underline{2} \ldots))$$

Manage environment explicitly using de Bruijn indices:

$$\underline{\lambda}((\underline{\lambda}(\mathtt{zero} + \mathtt{zero})) \\ (\mathtt{zero} + \text{出} \lambda k.\, (\underline{\lambda}(\mathtt{throw}\ (\mathtt{import}\ k)\ \mathtt{zero}))(\underline{40} + \underline{2})))$$

The continuation $k : [i : \mathsf{int}]\, \mathsf{int} \to []\, \mathsf{int}$

$\mathtt{import}\ k : [i : \mathsf{int}, x : \mathsf{int}]\, \mathsf{int} \to [x : \mathsf{int}]\, \mathsf{int}$

(Nanevski, Pfenning & Pientka 2008)

MetaOCaml today!

Environment classifiers

Judgments: $e : \tau \qquad \alpha / \tau_0 \qquad \alpha \leq \beta$

$$\frac{\alpha/\tau_0}{\underline{n} : \langle \mathsf{int} \rangle^\alpha} \qquad \frac{e_1 : \langle \mathsf{int} \rangle^\alpha \quad e_2 : \langle \mathsf{int} \rangle^\alpha}{e_1 + e_2 : \langle \mathsf{int} \rangle^\alpha}$$

$$\frac{e_1 : \langle v_1 \to v \rangle^\alpha \quad e_2 : \langle v_1 \rangle^\alpha}{e_1 e_2 : \langle v \rangle^\alpha} \qquad \frac{\begin{array}{c}[x : \langle v_1 \rangle^\alpha] \\ \vdots \\ e : (\langle v \rangle^\alpha \to \tau_0) \to \tau_0\end{array} \quad \alpha/\tau_0}{\underline{\lambda} x.\, e : (\langle v_1 \to v \rangle^\alpha \to \tau_0) \to \tau_0}$$

$$\frac{}{0/\mathsf{String}} \qquad \frac{\begin{array}{c}[\alpha \leq \beta \quad \beta/\tau_0] \\ \vdots \\ e : (\langle v \rangle^\beta \to \tau_0) \to \tau_0\end{array}}{\mathsf{region}\ e : (\langle v \rangle^\alpha \to \tau_0) \to \tau_0} \qquad \frac{e : \langle \tau \rangle^\alpha \quad \alpha \leq \beta}{e : \langle \tau \rangle^\beta}$$

Environment classifiers

Judgments: $e : \tau$ $\quad \alpha/\tau_0 \quad$ $\alpha \leq \beta$

$$\frac{\alpha/\tau_0}{\underline{n} : \langle \mathsf{int} \rangle^\alpha} \qquad \frac{e_1 : \langle \mathsf{int} \rangle^\alpha \quad e_2 : \langle \mathsf{int} \rangle^\alpha}{e_1 + e_2 : \langle \mathsf{int} \rangle^\alpha}$$

$$\frac{e_1 : \langle v_1 \to v \rangle^\alpha \quad e_2 : \langle v_1 \rangle^\alpha}{e_1 e_2 : \langle v \rangle^\alpha} \qquad \frac{\begin{array}{c}[x : \langle v_1 \rangle^\alpha] \\ \vdots \\ e : (\langle v \rangle^\alpha \to \tau_0) \to \tau_0\end{array} \quad \alpha/\tau_0}{\underline{\lambda} x.\, e : (\langle v_1 \to v \rangle^\alpha \to \tau_0) \to \tau_0}$$

$$\frac{}{0/\mathsf{String}} \qquad \frac{\begin{array}{c}[\alpha \leq \beta \quad \beta/\tau_0] \\ \vdots \\ e : (\langle v \rangle^\beta \to \tau_0) \to \tau_0\end{array}}{\mathsf{region}\ e : (\langle v \rangle^\alpha \to \tau_0) \to \tau_0} \qquad \frac{e : \langle \tau \rangle^\alpha \quad \alpha \leq \beta}{e : \langle \tau \rangle^\beta}$$

Environment classifiers

Judgments: $e : \tau$ $\quad \alpha/\tau_0$ $\quad \alpha \leq \beta$

$$\frac{\alpha/\tau_0}{\underline{n} : \langle \mathsf{int} \rangle^\alpha} \qquad \frac{e_1 : \langle \mathsf{int} \rangle^\alpha \quad e_2 : \langle \mathsf{int} \rangle^\alpha}{e_1 + e_2 : \langle \mathsf{int} \rangle^\alpha}$$

$$\frac{e_1 : \langle v_1 \to v \rangle^\alpha \quad e_2 : \langle v_1 \rangle^\alpha}{e_1 e_2 : \langle v \rangle^\alpha} \qquad \frac{\begin{array}{c}[x : \langle v_1 \rangle^\alpha] \\ \vdots \\ e : (\langle v \rangle^\alpha \to \tau_0) \to \tau_0\end{array} \quad \alpha/\tau_0}{\underline{\lambda} x.\, e : (\langle v_1 \to v \rangle^\alpha \to \tau_0) \to \tau_0}$$

$$\frac{}{0/\mathsf{String}} \qquad \frac{\begin{array}{c}[\alpha \leq \beta \quad \beta/\tau_0] \\ \vdots \\ e : (\langle v \rangle^\beta \to \tau_0) \to \tau_0\end{array}}{\mathsf{region}\ e : (\langle v \rangle^\alpha \to \tau_0) \to \tau_0} \qquad \frac{e : \langle \tau \rangle^\alpha \quad \alpha \leq \beta}{e : \langle \tau \rangle^\beta}$$

Environment classifiers

Judgments: $e : \tau$ $\qquad \alpha/\tau_0$ $\qquad \alpha \le \beta$

$$\frac{\alpha/\tau_0}{\underline{n} : \langle \mathsf{int} \rangle^\alpha} \qquad \frac{e_1 : \langle \mathsf{int} \rangle^\alpha \quad e_2 : \langle \mathsf{int} \rangle^\alpha}{e_1 + e_2 : \langle \mathsf{int} \rangle^\alpha}$$

$$\frac{e_1 : \langle v_1 \to v \rangle^\alpha \quad e_2 : \langle v_1 \rangle^\alpha}{e_1 e_2 : \langle v \rangle^\alpha} \qquad \frac{\begin{array}{c}[x : \langle v_1 \rangle^\alpha] \\ \vdots \\ e : (\langle v \rangle^\alpha \to \tau_0) \to \tau_0\end{array} \quad \alpha/\tau_0}{\underline{\lambda} x.\, e : (\langle v_1 \to v \rangle^\alpha \to \tau_0) \to \tau_0}$$

$$\frac{}{0/\mathsf{String}} \qquad \frac{\begin{array}{c}[\alpha \le \beta \quad \beta/\tau_0] \\ \vdots \\ e : (\langle v \rangle^\beta \to \tau_0) \to \tau_0\end{array}}{\mathsf{region}\ e : (\langle v \rangle^\alpha \to \tau_0) \to \tau_0} \qquad \frac{e : \langle \tau \rangle^\alpha \quad \alpha \le \beta}{e : \langle \tau \rangle^\beta}$$

Environment classifiers

Judgments: $e : \tau$ $\quad \alpha / \tau_0$ $\quad \alpha \leq \beta$

$$\frac{\alpha / \tau_0}{\underline{n} : \langle \mathsf{int} \rangle^\alpha} \qquad \frac{e_1 : \langle \mathsf{int} \rangle^\alpha \quad e_2 : \langle \mathsf{int} \rangle^\alpha}{e_1 + e_2 : \langle \mathsf{int} \rangle^\alpha}$$

$$\frac{e_1 : \langle \upsilon_1 \to \upsilon \rangle^\alpha \quad e_2 : \langle \upsilon_1 \rangle^\alpha}{e_1 e_2 : \langle \upsilon \rangle^\alpha} \qquad \frac{\begin{array}{c}[x : \langle \upsilon_1 \rangle^\alpha] \\ \vdots \\ e : (\langle \upsilon \rangle^\alpha \to \tau_0) \to \tau_0 \quad \alpha / \tau_0\end{array}}{\underline{\lambda} x.\, e : (\langle \upsilon_1 \to \upsilon \rangle^\alpha \to \tau_0) \to \tau_0}$$

$$\frac{}{0 / \mathsf{String}} \qquad \frac{\begin{array}{c}[\alpha \leq \beta \quad \beta / \tau_0] \\ \vdots \\ e : (\langle \upsilon \rangle^\beta \to \tau_0) \to \tau_0\end{array}}{\mathsf{region}\ e : (\langle \upsilon \rangle^\alpha \to \tau_0) \to \tau_0} \qquad \frac{e : \langle \tau \rangle^\alpha \quad \alpha \leq \beta}{e : \langle \tau \rangle^\beta}$$

Using environment classifiers

In CPS:

$$(\underline{\lambda} i.\, \lambda k.\, (\underline{\lambda} y.\, \lambda k'.\, k'(y+y))(\lambda l.\, (\ldots \underline{40} + \underline{2} \ldots)(\lambda x.\, k(l(i+x)))))\,(\lambda z.\, z)$$

Create a region for $\underline{\lambda} i$.

$$\text{region } (\underline{\lambda} i.\, \lambda k.\, (\underline{\lambda} y.\, \lambda k'.\, k'(y+y))(\lambda l.\, (\lambda k.\, \lambda m.\, \lambda n.\, (\underline{\lambda} x.\, kxm)(\lambda l.\, n(l(\underline{40}+\underline{2}))))(\lambda x.\, k(l(i+x)))))\,(\lambda z.\, \lambda k.\, kz)(\lambda z.\, \lambda k.\, kz)$$

Continuation hierarchy: k up to $\underline{\lambda} i$. m up to $\underline{\lambda} x$. n beyond

OCaml today!

The ends

Metalanguages for

- high-performance/embedded computing
- natural-language semantics of scope and quotation

need

- **safety** $\leftarrow$ track object variable bindings
- **clarity** $\leftarrow$ provide delimited control operators

Help!