Semantyka i weryfikacja programów

Bartosz Klin

(slajdy Andrzeja Tarleckiego)

Instytut Informatyki
Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski

`http://www.mimuw.edu.pl/~klin` pok. 5680
`klin@mimuw.edu.pl`

Strona tego wykładu:

`http://www.mimuw.edu.pl/~klin/sem18-19.html`

Program Semantics & Verification

Bartosz Klin

(slides courtesy of Andrzej Tarlecki)

Institute of Informatics
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw

`http://www.mimuw.edu.pl/~klin` office: 5680
`klin@mimuw.edu.pl`

This course: `http://www.mimuw.edu.pl/~klin/sem18-19.html`

Input/output

$\textsc{Tiny}^{+++}$

$$S \in \textbf{Stmt} ::= \ldots \mid \textbf{read}\ x \mid \textbf{write}\ e$$

Semantic domains

$$\begin{aligned}
\mathbf{Stream} &= \mathbf{Int} \times \mathbf{Stream} + \{\mathbf{eof}\}\\
\mathbf{Input} &= \mathbf{Stream}\\
\mathbf{Output} &= \mathbf{Stream}\\
\mathbf{State} &= \mathbf{Store} \times \mathbf{Input} \times \mathbf{Output}
\end{aligned}$$

Actually:

$$\mathbf{Stream} = (\mathbf{Int} \otimes_L \mathbf{Stream}) \oplus \{\mathbf{eof}\}_\perp$$

where:

$$\mathbf{D} \otimes_L \mathbf{D}' = \mathbf{D} \otimes \mathbf{D}'_\perp$$

Interpretation:

$$\mathbf{Stream} = \mathbf{Int}^* + \mathbf{Int}^\omega$$

Semantic functions

$$\mathcal{E} : \mathbf{Exp} \to \underbrace{\mathbf{VEnv} \to \mathbf{State} \to (\mathbf{Int} + \{??\})}_{\mathbf{EXP}}$$

$$\mathcal{B} : \mathbf{BExp} \to \underbrace{\mathbf{VEnv} \to \mathbf{State} \to (\mathbf{Bool} + \{??\})}_{\mathbf{BEXP}}$$

Only one clause to modify here:

$$\boxed{\mathcal{E}[\![x]\!]\ \rho_V\ \langle s, i, o\rangle = s\ l \ \text{where}\ l = \rho_V\ x}$$

Semantics of statements

$$\mathcal{S} : \mathbf{Stmt} \to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{State} \to (\mathbf{State} + \{??\})}_{\mathbf{STMT}}$$

Again, only one clause to change:

$$\mathcal{S}[\![x := e]\!]\ \rho_V\ \rho_P\ \langle s, i, o\rangle = \langle s[l \mapsto n], i, o\rangle \text{ where } l = \rho_V\ x, n = \mathcal{E}[\![e]\!]\ \rho_V\ \langle s, i, o\rangle$$

(plus a similar change in $\mathcal{D}_V[\![\mathbf{var}\ x; D_V]\!]\ \ldots = \ldots$) and two clauses to add:

$$\mathcal{S}[\![\mathbf{read}\ x]\!]\ \rho_V\ \rho_P\ \langle s, i, o\rangle = \langle s[l \mapsto n], i', o\rangle \text{ where } l = \rho_V\ x, \langle n, i'\rangle = i$$

$$\mathcal{S}[\![\mathbf{write}\ e]\!]\ \rho_V\ \rho_P\ \langle s, i, o\rangle = \langle s, i, \langle n, o\rangle\rangle \text{ where } n = \mathcal{E}[\![e]\!]\ \rho_V\ \langle s, i, o\rangle$$

Cheating a bit: writing out in the reverse order

Programs

New syntactic domain:

$$\boxed{\mathbf{Prog} ::= \mathbf{prog}\ S}$$

with obvious semantic function:

$$\boxed{\mathcal{P} : \mathbf{Prog} \to \underbrace{\mathbf{Input} \to (\mathbf{Output} + \{??\})}_{\mathbf{PROG}}}$$

given by:

$$\boxed{\begin{array}{l} \mathcal{P}[\![\mathbf{prog}\ S]\!]\ i = o' \text{ where } \mathcal{S}[\![S]\!]\ \rho_V^{\emptyset}\ \rho_P^{\emptyset}\ \langle s^{\emptyset}, i, \mathbf{eof}\rangle = \langle s', i', o'\rangle, \\ \qquad\qquad\qquad\qquad \rho_V^{\emptyset}\ x = ??, \rho_P^{\emptyset}\ p = ??, s^{\emptyset}\ next = 0, s^{\emptyset}\ l = ?? \end{array}}$$

Okay, but. . .

Do we want to allow statements to erase output?

Changing philosophy

From: What happens now?

To: What the overall answer will be?

Direct semantics

begin … ; … ; … **end**

$$s^{\emptyset} \xrightarrow{\mathcal{S}[\![\dots]\!]} s_i \xrightarrow{\mathcal{S}[\![\dots]\!]} s_j \xrightarrow{\mathcal{S}[\![\dots]\!]} s' \rightsquigarrow \text{“overall result”}$$

Continuation semantics

begin … ; … ; … **end**

$$\kappa' : \rightsquigarrow \text{“overall result”}$$

$$\xleftarrow{\mathcal{S}[\![\dots]\!]} \kappa_i : \rightsquigarrow \text{“overall result”}$$

$$\xleftarrow{\mathcal{S}[\![\dots]\!]} \kappa_j : \rightsquigarrow \text{“overall result”}$$

$$\xleftarrow{\mathcal{S}[\![\dots]\!]} \kappa^{\emptyset} : \rightsquigarrow \text{“overall result”}$$

Continuations

$$\boxed{\mathbf{Cont} = \mathbf{State} \to \mathbf{Ans}}$$

Now:

$$\boxed{\begin{aligned}\mathbf{State} &= \mathbf{Store} \times \mathbf{Input} \\ \mathbf{Ans} &= \mathbf{Output}\end{aligned}}$$

- states do not include outputs
- answers are outputs (or errors)
- these are continuations for statements; semantics for statements is given by:

$$\boxed{\mathcal{S} : \mathbf{Stmt} \to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{Cont} \to \mathbf{Cont}}_{\mathbf{STMT}}}$$

That is: $\boxed{\mathbf{STMT} = \mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{Cont} \to \mathbf{State} \to \mathbf{Ans}}$

Expression and declaration continuations

- continuations for other syntactic categories should be additionally parameterised by whatever these pass on:
  - expressions pass on values, so

$$\boxed{\begin{array}{l}\mathbf{Cont_E} = \mathbf{Int} \to \mathbf{State} \to \mathbf{Ans}\\ \mathbf{Cont_B} = \mathbf{Bool} \to \mathbf{State} \to \mathbf{Ans}\end{array}}$$

  - declarations pass on environments, so

$$\boxed{\begin{array}{l}\mathbf{Cont_{D_V}} = \mathbf{VEnv} \to \mathbf{State} \to \mathbf{Ans}\\ \mathbf{Cont_{D_P}} = \mathbf{PEnv} \to \mathbf{State} \to \mathbf{Ans}\end{array}}$$

$\textsc{Tiny}^{+++}$

$$
\begin{aligned}
N \in \mathbf{Num} &::= 0 \mid 1 \mid 2 \mid \cdots \\
x \in \mathbf{Var} &::= \cdots \\
p \in \mathbf{IDE} &::= \cdots \\
e \in \mathbf{Exp} &::= N \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2 \\
b \in \mathbf{BExp} &::= \mathbf{true} \mid \mathbf{false} \mid e_1 \leq e_2 \mid \neg b' \mid b_1 \wedge b_2 \\
S \in \mathbf{Stmt} &::= x := e \mid \mathbf{skip} \mid S_1 ; S_2 \mid \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \mid \mathbf{while}\ b\ \mathbf{do}\ S' \\
&\quad\mid \mathbf{begin}\ D_V\ D_P\ S\ \mathbf{end} \mid \mathbf{call}\ p \mid \mathbf{call}\ p(\mathbf{vr}\ x) \\
&\quad\mid \mathbf{read}\ x \mid \mathbf{write}\ e \\
D_V \in \mathbf{VDecl} &::= \mathbf{var}\ x; D_V \mid \varepsilon \\
D_P \in \mathbf{PDecl} &::= \mathbf{proc}\ p\ \mathbf{is}\ (S); D_P \mid \mathbf{proc}\ p(\mathbf{vr}\ x)\ \mathbf{is}\ (S); D_P \mid \varepsilon \\
\mathbf{Prog} &::= \mathbf{prog}\ S
\end{aligned}
$$

Semantic domains

$$\begin{aligned} \mathbf{Int} &= \ldots \\ \mathbf{Bool} &= \ldots \\ \mathbf{Loc} &= \ldots \\ \mathbf{Store} &= \ldots \\ \mathbf{VEnv} &= \ldots \end{aligned}$$

$$\begin{aligned} \mathbf{Input} &= \mathbf{Int} \times \mathbf{Input} + \{\mathbf{eof}\} \\ \mathbf{State} &= \mathbf{Store} \times \mathbf{Input} \\ \mathbf{Output} &= \mathbf{Int} \times \mathbf{Output} + \{\mathbf{eof}, ??\} \end{aligned}$$

$$\begin{aligned} \mathbf{Cont} &= \mathbf{State} \to \mathbf{Output} \\ \mathbf{Cont_E} &= \mathbf{Int} \to \mathbf{Cont} \\ \mathbf{Cont_B} &= \mathbf{Bool} \to \mathbf{Cont} \\ \mathbf{Cont_{D_V}} &= \mathbf{VEnv} \to \mathbf{Cont} \\ \mathbf{Cont_{D_P}} &= \mathbf{PEnv} \to \mathbf{Cont} \end{aligned}$$

$$\begin{aligned} \mathbf{PROC}_0 &= \mathbf{Cont} \to \mathbf{Cont} \\ \mathbf{PROC}_1^{\mathsf{vr}} &= \mathbf{Loc} \to \mathbf{PROC}_0 \\ \mathbf{PEnv} &= \\ &\quad \mathbf{IDE} \to (\mathbf{PROC}_0 + \mathbf{PROC}_1^{\mathsf{vr}} + \{??\}) \end{aligned}$$

Semantic functions

$$\mathcal{E}: \mathbf{Exp} \to \underbrace{\mathbf{VEnv} \to \mathbf{Cont_E} \to \mathbf{Cont}}_{\mathbf{EXP}}$$

$$\mathcal{B}: \mathbf{BExp} \to \underbrace{\mathbf{VEnv} \to \mathbf{Cont_B} \to \mathbf{Cont}}_{\mathbf{BEXP}}$$

$$\mathcal{S}: \mathbf{Stmt} \to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{Cont} \to \mathbf{Cont}}_{\mathbf{STMT}}$$

$$\mathcal{D}_V: \mathbf{VDecl} \to \underbrace{\mathbf{VEnv} \to \mathbf{Cont_{D_V}} \to \mathbf{Cont}}_{\mathbf{VDECL}}$$

$$\mathcal{D}_P: \mathbf{PDecl} \to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{Cont_{D_P}} \to \mathbf{Cont}}_{\mathbf{PDECL}}$$

$$\mathcal{P}: \mathbf{Prog} \to \underbrace{\mathbf{Input} \to \mathbf{Output}}_{\mathbf{PROG}}$$

Sample semantic clauses

Programs:

$$\mathcal{P}[\![\textbf{prog}\ S]\!]\ i = \mathcal{S}[\![S]\!]\ \rho_V^{\emptyset}\ \rho_P^{\emptyset}\ \kappa^{\emptyset}\ \langle s^{\emptyset}, i\rangle$$
$$\text{where } \rho_V^{\emptyset}\ x = ??, \rho_P^{\emptyset}\ p = ??, \kappa^{\emptyset}\ s = \textbf{eof}, s^{\emptyset}\ next = 0, s^{\emptyset}\ l = ??$$

Declarations:

$$\mathcal{D}_P[\![\varepsilon]\!]\ \rho_V\ \rho_P\ \kappa_P = \kappa_P\ \rho_P$$
$$\mathcal{D}_P[\![\textbf{proc}\ p\ \textbf{is}\ (S); D_P]\!]\ \rho_V\ \rho_P\ =$$
$$\mathcal{D}_P[\![D_P]\!]\ \rho_V\ \rho_P[p \mapsto P]\ \text{where } P = \mathcal{S}[\![S]\!]\ \rho_V\ \rho_P[p \mapsto P]$$
$$\mathcal{D}_V[\![\textbf{var}\ x; D_V]\!]\ \rho_V\ \kappa_V\ \langle s, i\rangle =$$
$$\mathcal{D}_V[\![D_V]\!]\ \rho_V'\ \kappa_V\ \langle s', i\rangle\ \text{where } l = s\ next, \rho_V' = \rho_V[x \mapsto l],$$
$$s' = s[l \mapsto ??, next \mapsto l + 1]$$

No continuations really used here, but: may be rewritten to a more standard continuation style

Sample semantic clauses

Expressions:

$$\mathcal{E}[\![x]\!]\ \rho_V\ \kappa_E = \lambda\langle s,i\rangle{:}\mathbf{State}.\kappa_E\ n\ \langle s,i\rangle \quad \text{where } l = \rho_V\ x, n = s\ l$$

this means: ?? if $\rho_V\ x = ??$ or $s\ l = ??$

$$\mathcal{E}[\![e_1 + e_2]\!]\ \rho_V\ \kappa_E =$$
$$\mathcal{E}[\![e_1]\!]\ \rho_V\ \lambda n_1{:}\mathbf{Int}.\mathcal{E}[\![e_2]\!]\ \rho_V\ \lambda n_2{:}\mathbf{Int}.\kappa_E\ (n_1 + n_2)$$

check the types!

Boolean expressions:

$$\mathcal{B}[\![\mathbf{true}]\!]\ \rho_V\ \kappa_B = \kappa_B\ \mathbf{tt}$$

$$\mathcal{B}[\![e_1 \leq e_2]\!]\ \rho_V\ \kappa_B =$$
$$\mathcal{E}[\![e_1]\!]\ \rho_V\ \lambda n_1{:}\mathbf{Int}.\mathcal{E}[\![e_2]\!]\ \rho_V\ \lambda n_2{:}\mathbf{Int}.$$
$$\kappa_B\ \mathit{ifte}(n_1 \leq n_2, \mathbf{tt}, \mathbf{ff})$$

Back to declarations

Recall:

$$\begin{aligned}&\mathcal{D}_V[\![\mathbf{var}\ x; D_V]\!]\ \rho_V\ \kappa_V\ \langle s, i\rangle = \\ &\quad \mathcal{D}_V[\![D_V]\!]\ \rho'_V\ \kappa_V\ \langle s', i\rangle \text{ where } l = s\ \mathit{next}, \rho'_V = \rho_V[x \mapsto l], \\ &\qquad\qquad\qquad\qquad s' = s[l \mapsto ??, \mathit{next} \mapsto l+1]\end{aligned}$$

What would happen if variable declarations included initializing expressions?

$$\begin{aligned}&\mathcal{D}_V[\![\mathbf{var}\ x = e; D_V]\!]\ \rho_V\ \kappa_V\ \langle s, i\rangle = \\ &\quad \mathcal{E}[\![e]\!]\ \rho_V\ (\lambda n{:}\mathbf{Int}.\mathcal{D}_V[\![D_V]\!]\ \rho'_V\ \kappa_V\ \langle s', i\rangle) \text{ where } l = s\ \mathit{next}, \rho'_V = \rho_V[x \mapsto l], \\ &\qquad\qquad\qquad\qquad s' = s[l \mapsto n, \mathit{next} \mapsto l+1]\end{aligned}$$

Statements

$$\mathcal{S}\llbracket x := e\rrbracket\ \rho_V\ \rho_P\ \kappa = \mathcal{E}\llbracket e\rrbracket\ \rho_V\ (\lambda n{:}\mathbf{Int}.\lambda\langle s, i\rangle{:}\mathbf{State}.\kappa\ \langle s[l \mapsto n], i\rangle)$$

$$\text{where } l = \rho_V\ x$$

$$\mathcal{S}\llbracket \mathbf{skip}\rrbracket\ \rho_V\ \rho_P = id_{\mathbf{Cont}}$$

$$\mathcal{S}\llbracket S_1; S_2\rrbracket\ \rho_V\ \rho_P\ \kappa = \mathcal{S}\llbracket S_1\rrbracket\ \rho_V\ \rho_P\ (\mathcal{S}\llbracket S_2\rrbracket\ \rho_V\ \rho_P\ \kappa)$$

$$\mathcal{S}\llbracket \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\rrbracket\ \rho_V\ \rho_P\ \kappa =$$
$$\qquad \mathcal{B}\llbracket b\rrbracket\ \rho_V\ \lambda v{:}\mathbf{Bool}.\mathit{ifte}(v, \mathcal{S}\llbracket S_1\rrbracket\ \rho_V\ \rho_P\ \kappa, \mathcal{S}\llbracket S_2\rrbracket\ \rho_V\ \rho_P\ \kappa)$$

$$\mathcal{S}\llbracket \mathbf{while}\ b\ \mathbf{do}\ S\rrbracket\ \rho_V\ \rho_P\ \kappa =$$
$$\qquad \mathcal{B}\llbracket b\rrbracket\ \rho_V\ \lambda v{:}\mathbf{Bool}.\mathit{ifte}(v, \mathcal{S}\llbracket S\rrbracket\ \rho_V\ \rho_P\ (\mathcal{S}\llbracket \mathbf{while}\ b\ \mathbf{do}\ S\rrbracket\ \rho_V\ \rho_P\ \kappa), \kappa)$$

$$\mathcal{S}\llbracket \mathbf{call}\ p\rrbracket\ \rho_V\ \rho_P = P\ \text{ where }\ P = \rho_P\ p$$

$$\mathcal{S}\llbracket \mathbf{call}\ p(\mathbf{vr}\ x)\rrbracket\ \rho_V\ \rho_P = P\ l\ \text{ where }\ P = \rho_P\ p \in \mathbf{PROC}_1^{\mathsf{vr}},\ \ l = \rho_V\ x$$

$$\mathcal{S}\llbracket \mathbf{read}\ x\rrbracket\ \rho_V\ \rho_P\ \kappa\ \langle s, i\rangle = \kappa\ \langle s[l \mapsto n], i'\rangle\ \text{ where }\ l = \rho_V\ x, \langle n, i'\rangle = i$$

$$\mathcal{S}\llbracket \mathbf{write}\ e\rrbracket\ \rho_V\ \rho_P\ \kappa = \mathcal{E}\llbracket e\rrbracket\ \rho_V\ \lambda n{:}\mathbf{Int}.\lambda\langle s, i\rangle{:}\mathbf{State}.\langle n, \kappa\ \langle s, i\rangle\rangle$$

Blocks

$$\begin{aligned}&\mathcal{S}[\![\textbf{begin}\ D_V\ D_P\ S\ \textbf{end}]\!]\ \rho_V\ \rho_P\ \kappa =\\ &\qquad \mathcal{D}_V[\![D_V]\!]\ \rho_V\ \lambda\rho'_V{:}\mathbf{VEnv}.\mathcal{D}_P[\![D_P]\!]\ \rho'_V\ \rho_P\ \lambda\rho'_P{:}\mathbf{PEnv}.\mathcal{S}[\![S]\!]\ \rho'_V\ \rho'_P\ \kappa\end{aligned}$$

This got separated, because we will want to add jumps. . .

Abrupt termination

Let us forget input/output for now, fall back to the language $\textsc{Tiny}^{++}$with (paremeterless) procedures.

Extend it with:

$$S \in \mathbf{Stmt} ::= \ldots \mid \mathbf{abort}$$

The killer application of continuations is non-local control flow.

This is the simplest example.

Semantic domains

$$\begin{aligned}
\mathbf{Ans} &= \mathbf{Store}\\
\mathbf{Cont} &= \mathbf{Store} \to \mathbf{Ans}\\
\mathbf{Cont_E} &= \mathbf{Int} \to \mathbf{Ans}\\
\mathbf{Cont_B} &= \mathbf{Bool} \to \mathbf{Ans}\\
\mathbf{Cont_{D_V}} &= \mathbf{VEnv} \to \mathbf{Cont}\\
\mathbf{Cont_{D_P}} &= \mathbf{PEnv} \to \mathbf{Ans}\\
\mathbf{PROC} &= \mathbf{Cont} \to \mathbf{Cont}\\
\mathbf{PEnv} &= \mathbf{IDE} \to (\mathbf{PROC} + \{??\})
\end{aligned}$$

Most continuation types got simplified, since expressions or procedure declarations do not produce new $\mathbf{Store}$'s. We could have done that previously, too.

Semantic functions

As before:

$$
\begin{aligned}
\mathcal{E}: &\quad \mathbf{Exp} \to \underbrace{\mathbf{VEnv} \to \mathbf{Cont_E} \to \mathbf{Cont}}_{\mathbf{EXP}}\\
\mathcal{B}: &\quad \mathbf{BExp} \to \underbrace{\mathbf{VEnv} \to \mathbf{Cont_B} \to \mathbf{Cont}}_{\mathbf{BEXP}}\\
\mathcal{S}: &\quad \mathbf{Stmt} \to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{Cont} \to \mathbf{Cont}}_{\mathbf{STMT}}\\
\mathcal{D}_V: &\quad \mathbf{VDecl} \to \underbrace{\mathbf{VEnv} \to \mathbf{Cont_{D_V}} \to \mathbf{Cont}}_{\mathbf{VDECL}}\\
\mathcal{D}_P: &\quad \mathbf{PDecl} \to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{Cont_{D_P}} \to \mathbf{Cont}}_{\mathbf{PDECL}}
\end{aligned}
$$

Sample semantic clauses

Roughly as before. A few get simpler because of simpler continuation types, e.g.:

$$\boxed{\mathcal{E}[\![x]\!]\ \rho_V\ \kappa_E\ s = \kappa_E\ n \text{ where } l = \rho_V\ x, n = s\ l}$$

But a few get more complicated because the simpler types require more explicit state passing, e.g.:

$$\boxed{\mathcal{E}[\![e_1 + e_2]\!]\ \rho_V\ \kappa_E\ s = \mathcal{E}[\![e_1]\!]\ \rho_V\ (\lambda n_1{:}\mathbf{Int}.\mathcal{E}[\![e_2]\!]\ \rho_V\ (\lambda n_2{:}\mathbf{Int}.\kappa_E\ (n_1 + n_2))\ s)\ s}$$

One new clause:

$$\boxed{\mathcal{S}[\![\mathbf{abort}]\!]\ \rho_V\ \rho_P\ \kappa = id_{\mathbf{Store}}}$$

Compare it to:

$$\boxed{\mathcal{S}[\![\mathbf{skip}]\!]\ \rho_V\ \rho_P\ = id_{\mathbf{Cont}}}$$

Exceptions

Exception throwing is a more fancy kind of abrupt termination, where only part of a program gets terminated.

We will throw and catch named exceptions without parameters.

$$S \in \mathbf{Stmt} ::= \ldots \mid \mathbf{try}\ S_1\ \mathbf{catch}(\chi)\ S_2 \mid \mathbf{throw}\ \chi$$

$$\chi \in \mathbf{EXN} ::= \ldots$$

- A thrown exception may erase a part of the procedure-call stack, but it does not erase changes to the store.

Semantic domains

- A new kind of environment:

$$\boxed{\mathbf{XEnv} = \mathbf{EXN} \to (\mathbf{Cont} + \{??\})}$$

- The appropriate semantic functions get another environment parameter:

$$\boxed{\begin{aligned} \mathcal{S}:\quad \mathbf{Stmt} &\to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{XEnv} \to \mathbf{Cont} \to \mathbf{Cont}}_{\mathbf{STMT}} \\ \mathcal{D}_P: \mathbf{PDecl} &\to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{XEnv} \to \mathbf{Cont}_{\mathbf{D_P}} \to \mathbf{Cont}}_{\mathbf{PDECL}} \end{aligned}}$$

Semantic clauses

- Semantic clauses for declarations and statements of the “old” forms take the extra environment parameter and disregard it (passing it “down”).
- New clauses:

$$\mathcal{S}[\![\mathbf{try}\ S_1\ \mathbf{catch}(\chi)\ S_2]\!]\ \rho_V\ \rho_P\ \rho_X\ \kappa =$$

$$\qquad \mathcal{S}[\![S_1]\!]\ \rho_V\ \rho_P\ \rho_X[\chi \mapsto \kappa']\ \kappa \text{ where } \kappa' = \mathcal{S}[\![S_2]\!]\ \rho_V\ \rho_P\ \rho_X\ \kappa$$

$$\mathcal{S}[\![\mathbf{throw}\ \chi]\!]\ \rho_V\ \rho_P\ \rho_X\ \kappa = \rho_X\ \chi$$

Goto's

- Let's replace exceptions by the full control-flow catastrophe.

$$
\begin{aligned}
&S \in \mathbf{Stmt} ::= \ldots \mid L{:}S \mid \mathbf{goto}\ L\\
&L \in \mathbf{LAB} ::= \ldots
\end{aligned}
$$

- Labels are visible inside the block in which they are declared
- No jumps into a block are allowed; jumps into other statements are okay

Semantics — sketch

- Yet another environment:

$$\boxed{\mathbf{LEnv} = \mathbf{LAB} \to (\mathbf{Cont} + \{??\})}$$

- Semantic functions get another environment parameter as before:

$$\mathcal{S}: \quad \mathbf{Stmt} \to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{LEnv} \to \mathbf{Cont} \to \mathbf{Cont}}_{\mathbf{STMT}}$$

$$\mathcal{D}_P: \mathbf{PDecl} \to \underbrace{\mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{LEnv} \to \mathbf{Cont}_{\mathbf{D_P}} \to \mathbf{Cont}}_{\mathbf{PDECL}}$$

- Semantic clauses for declarations and statements of the "old" forms take the extra parameter and disregard it (passing it "down").

Goto's — sketch of the semantics continues

- We add a declaration-like semantics for statements:

$$\boxed{\mathcal{D}_S : \mathbf{Stmt} \to \mathbf{VEnv} \to \mathbf{PEnv} \to \mathbf{LEnv} \to \mathbf{Cont} \to \mathbf{LEnv}}$$

- With a few trivial clauses, like:

$$\boxed{\mathcal{D}_S[\![x := e]\!]\,\rho_V\ \rho_P\ \rho_L\ \kappa = \rho_L}$$

and similarly for **skip**, **call** p etc., and for **goto** L, where no visible labels can be introduced. Since we cannot jump into blocks, also:

$$\boxed{\mathcal{D}_S[\![\mathbf{begin}\ D_V\ \ D_P\ \ S\ \mathbf{end}]\!]\,\rho_V\ \rho_P\ \rho_L\ \kappa = \rho_L}$$

Goto's — sketch of the semantics continues

- And then a few not quite so trivial clauses follow:

$$
\begin{aligned}
&\mathcal{D}_S[\![S_1; S_2]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \\
&\qquad \mathcal{D}_S[\![S_1]\!]\ \rho_V\ \rho_P\ \rho_L\ (\mathcal{S}[\![S_2]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa) + \mathcal{D}_S[\![S_2]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa \\
&\mathcal{D}_S[\![\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \\
&\qquad \mathcal{D}_S[\![S_1]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa + \mathcal{D}_S[\![S_2]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa \\
&\mathcal{D}_S[\![\textbf{while}\ b\ \textbf{do}\ S]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \\
&\qquad \mathcal{D}_S[\![S]\!]\ \rho_V\ \rho_P\ \rho_L\ (\mathcal{S}[\![\textbf{while}\ b\ \textbf{do}\ S]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa) \\
&\mathcal{D}_S[\![L{:}S]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \\
&\qquad (\mathcal{D}_S[\![S]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa)[L \mapsto \mathcal{S}[\![S]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa]
\end{aligned}
$$

The only extra thing to explain here is "updating":

$$
(\rho_L + \rho'_L)\ L = \begin{cases} \rho_L\ L & \text{if } \rho'_L\ L = ?? \\ \rho'_L\ L & \text{if } \rho'_L\ L \neq ?? \end{cases}
$$

Goto's — sketch of the semantics continues

- And finally we need new clauses for the (usual) semantics of labelled statements, of jumps (trivial now) and of blocks — rather complicated:

$$\mathcal{S}[\![L{:}S]\!] = \mathcal{S}[\![S]\!]$$

$$\mathcal{S}[\![\mathbf{goto}\ L]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \kappa_L \text{ where } \kappa_L = \rho_L\ L$$

$$\begin{array}{l}\mathcal{S}[\![\mathbf{begin}\ D_V\ \ D_P\ \ S\ \mathbf{end}]\!]\ \rho_V\ \rho_P\ \rho_L\ \kappa = \\ \quad \mathcal{D}_V[\![D_V]\!]\ \rho_V\ \lambda\rho'_V{:}\mathbf{VEnv}.\mathcal{D}_P[\![D_P]\!]\ \rho'_V\ \rho_P\ \rho_L\ \lambda\rho'_P{:}\mathbf{PEnv}. \\ \qquad\qquad \mathcal{S}[\![S]\!]\ \rho'_V\ \rho'_P\ \rho'_L\ \kappa \text{ where } \rho'_L = \mathcal{D}_S[\![S]\!]\ \rho'_V\ \rho'_P\ (\rho_L + \rho'_L)\ \kappa\end{array}$$

. . . and perhaps not quite right?

- one should really check if the labels within a block are unique — this is easy!
- labels within a block should be visible within procedure declarations in this block

callcc

- short for **call-with-current-continuation**
- **goto** on steroids
- featured in **Scheme**, **SML**, **Haskell**, **Ruby**, but also in fancy libraries for **C++**
- We shall not define its semantics, as it does not mix very well with TINY, an imperative language with a trivial type system.
- Instead, we will explain how it works, rather informally and by example.
- This illustrates how knowledge of semantic concepts can help the programmer to learn a programming language concept.

Taxonomy of jumps

	Static (lexical)	Dynamic
Outward only	**break** **return**	**throw** **setjmp()/longjmp()**
Arbitrary	**goto**	**callcc**

callcc explained

- **callcc** is a function that takes one argument.
- As a programmer, you typically need to prepare such an argument (call it **f**).
- Your **f** should be a function that takes one argument (call it **k**).
- Normally you never need to prepare **k**; your **f** should be prepared to get one.
- You can expect **k** to be a function that takes one argument.
- If you call **callcc f**:
  - **f** is called with a magical argument **k**
  - if at some point you call **k** with an argument **v**, then **f** is immediately terminated and **callcc f** takes value **v**.
  - if **f** terminates normally and returns a value **v**, then **callcc f** takes value **v**.
- *The above works even if* **f** *returns* **k***!* If **k** is called after exiting from **f**, the call stack may need to be rebuilt.

Examples (in a functional-like pseudocode)

```
let f k = k 42
in callcc f
```

evaluates to 42

```
let f k = (k 42) + 25
in callcc f
```

evaluates to 42

```
let f k =
  for x in L do
    if test x then k x
in callcc f
```

returns the first element of L for which `test` holds

```
z := ...;
let f k = (z := k;  1)
in (callcc f) + 1;
```

returns 2 and stores the "+1" function in z

The yin-yang puzzle

Even the author of this program could not understand why it does what it does.

```
let
  yin  = ((\c -> (print 0) c) (callcc id))
  yang = ((\c -> (print 1) c) (callcc id))
in
  (yin yang)
```

So, what does this thing do and why?

“Standard semantics”

- continuations (to handle jumps of various kinds, and simplify notation)
- careful classification of various domains of values (assignable, storable, output-able, closures, etc) with the corresponding semantics of expressions (of various kinds)
- Scott domains and domain equations
- continuous functions only
- . . .

. . . to be explained . . .