Operational Semantics

Jan Midtgaard

Winter School, Day 2

http://janmidtgaard.dk/aiws15/

Saint Petersburg, Russia, 2015

Yesterday

- Mathematical basis:
  - Transition systems
  - Partially ordered sets (posets), Complete partial orders (CPOs), Complete lattices
  - Galois connections
  - Fixed points
- Abstract interpretation basics:
  - Reachable states collecting semantics
  - Galois-connection based abstract interpretation
  - The alternative widening/narrowing framework
- OCaml intro

Semantics

Semantics according to Merriam-Webster

Main Entry: se·man·tics
Pronunciation: si-’man-tiks
Function: noun plural but singular or plural in construction
Date: 1893

1. *the study of meanings*: a : the historical and psychological study and the classification of changes in the signification of words or forms viewed as factors in linguistic development b (1) : semiotic (2) : a branch of semiotic dealing with the relations between signs and what they refer to and including theories of denotation, extension, naming, and truth
2. general semantics
3. a : *the meaning or relationship of meanings* of a sign or set of signs; especially : connotative meaning b : the language used (as in advertising or political propaganda) to achieve a desired effect on an audience especially through the use of words with novel or dual meanings

Semantics in Computer Science

Semantics is concerned with constructing formal models or specifications of systems. Examples of such systems include: Java, ML, JavaScript, . . . , JVM, x86, . . .

A model in itself is useful

- to understand features (scope, exceptions, continuations,...)
- to prove equivalence of programs
- to prove program transformations correct
- to prove properties (e.g., type safety)

In this course semantics will be the starting point for abstraction/approximation.

Many forms of semantics

- Denotational semantics
- Operational semantics
  - abstract machines/transition systems
  - structured operational semantics
  - big-step/natural/relational semantics
- Reduction semantics
- Axiomatic semantics/Hoare logic
- Game semantics

Hence enough for a separate course.

Semantics in this course

In this course we will focus on abstract machines, i.e., transition systems. These models are *operational* in that they describe the inner workings of an idealized machine.

Today we'll study semantics of four different languages:

- of three counter machine programs
- of CPS programs
- of IMP programs
- of bytecode programs

Throughout we take the AST view: We assume that all ambiguities have been resolved, and we will work with (and reason about) programs as abstract syntax trees.

Warm-up: The three counter machine

Plotkin's three counter machine (1/2)

There are 3 variables (or registers):

$$var \in Var = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \qquad \text{(variables)}$$

$$\begin{aligned} Inst ::= &\ \mathtt{inc}\ var \\ |&\ \mathtt{dec}\ var \\ |&\ \mathtt{zero}\ var\ pc\ \mathtt{else}\ pc' \\ |&\ \mathtt{stop} \end{aligned} \qquad \text{(instructions)}$$

$$P = Inst^* \qquad \text{(programs)}$$

$$pc \in PC = \mathbb{N} \qquad \text{(program counter)}$$

$$States = PC \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \qquad \text{(states)}$$

Initial state: $\langle 1,\ i,\ 0,\ 0\rangle$ (for program P with input i)

Final state: $\langle pc,\ 0,\ yv,\ 0\rangle$
(with yv being the result and where $P_{pc} = \mathtt{stop}$)

Plotkin's three counter machine (1/2)

There are 3 variables (or registers):

$$var \in Var = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \qquad \text{(variables)}$$

$$\begin{aligned} Inst ::= & \ \mathtt{inc}\ var \\ | & \ \mathtt{dec}\ var \\ | & \ \mathtt{zero}\ var\ pc\ \mathtt{else}\ pc' \\ | & \ \mathtt{stop} \end{aligned} \qquad \text{(instructions)}$$

$$P = Inst^* \qquad \text{(programs)}$$

$$pc \in PC = \mathbb{N} \qquad \text{(program counter)}$$

$$States = PC \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \qquad \text{(states)}$$

Initial states: $\{\langle 1,\ i,\ 0,\ 0\rangle \mid i \in \mathbb{N}_0\}$ (for program P with input i)

Final states: $\{\langle pc,\ 0,\ yv,\ 0\rangle \mid pc \in PC\ \wedge\ yv \in \mathbb{N}_0\ \wedge\ P_{pc} = \mathtt{stop}\}$ (with yv being the result)

Plotkin's three counter machine (2/2)

Transition relation:

$$
\begin{array}{lcll}
\langle pc,\, xv,\, yv,\, zv\rangle & \longrightarrow & \langle pc+1,\, xv+1,\, yv,\, zv\rangle & \text{if } P_{pc} = \texttt{inc x} \\
- & \longrightarrow & \langle pc+1,\, xv,\, yv+1,\, zv\rangle & \text{if } P_{pc} = \texttt{inc y} \\
- & \longrightarrow & \langle pc+1,\, xv,\, yv,\, zv+1\rangle & \text{if } P_{pc} = \texttt{inc z} \\
\\
\langle pc,\, xv,\, yv,\, zv\rangle & \longrightarrow & \langle pc+1,\, xv-1,\, yv,\, zv\rangle & \text{if } P_{pc} = \texttt{dec x} \ \wedge\ xv > 0 \\
- & \longrightarrow & \langle pc+1,\, xv,\, yv-1,\, zv\rangle & \text{if } P_{pc} = \texttt{dec y} \ \wedge\ yv > 0 \\
- & \longrightarrow & \langle pc+1,\, xv,\, yv,\, zv-1\rangle & \text{if } P_{pc} = \texttt{dec z} \ \wedge\ zv > 0 \\
\\
\langle pc,\, xv,\, yv,\, zv\rangle & \longrightarrow & \langle pc',\, xv,\, yv,\, zv\rangle & \text{if } P_{pc} = \texttt{zero x}\ pc'\ \texttt{else}\ pc'' \ \wedge\ xv = 0 \\
- & \longrightarrow & \langle pc'',\, xv,\, yv,\, zv\rangle & \text{if } P_{pc} = \texttt{zero x}\ pc'\ \texttt{else}\ pc'' \ \wedge\ xv \neq 0 \\
\\
\langle pc,\, xv,\, yv,\, zv\rangle & \longrightarrow & \langle pc',\, xv,\, yv,\, zv\rangle & \text{if } P_{pc} = \texttt{zero y}\ pc'\ \texttt{else}\ pc'' \ \wedge\ yv = 0 \\
- & \longrightarrow & \langle pc'',\, xv,\, yv,\, zv\rangle & \text{if } P_{pc} = \texttt{zero y}\ pc'\ \texttt{else}\ pc'' \ \wedge\ yv \neq 0 \\
\\
\langle pc,\, xv,\, yv,\, zv\rangle & \longrightarrow & \langle pc',\, xv,\, yv,\, zv\rangle & \text{if } P_{pc} = \texttt{zero z}\ pc'\ \texttt{else}\ pc'' \ \wedge\ zv = 0 \\
- & \longrightarrow & \langle pc'',\, xv,\, yv,\, zv\rangle & \text{if } P_{pc} = \texttt{zero z}\ pc'\ \texttt{else}\ pc'' \ \wedge\ zv \neq 0
\end{array}
$$

Note: there is no case for the `stop` instruction.

Also note: this version differs slightly from Plotkin's.

Exercise

Compute the first five execution steps of the following program for input 1:

```
1   zero x 6 else 2
2   dec x
3   inc y
4   inc y
5   zero x 6 else 2
6   stop
```

Exercise

Compute the first five execution steps of the following program for input 1:

```
1   zero x 6 else 2
2   dec x
3   inc y
4   inc y
5   zero x 6 else 2
6   stop
```

Bonus question: how can we encode unconditional jumps?

CPS semantics

Representing functional values

In languages like JavaScript, Scheme, and ML functions are first class values. That means the result of:

```
((λ (x) (λ (y) (+ x y))) 3)
```

is a functional value `(λ (y) (+ x y))` in which `x` is bound to 3.

Representing functional values

In languages like JavaScript, Scheme, and ML functions are first class values. That means the result of:

$$\texttt{((}\lambda\texttt{ (x) (}\lambda\texttt{ (y) (+ x y))) 3)}$$

is a functional value $\texttt{(}\lambda\texttt{ (y) (+ x y))}$ in which $\texttt{x}$ is bound to 3.

To represent such a value we could substitute all free occurrences of $\texttt{x}$ with 3. Alternatively we can record substitutions in an *environment* and represent functional values as $lambda \times env$- pairs:

$$\langle\texttt{(}\lambda\texttt{ (y) (+ x y))}, \bullet[x \mapsto 3]\rangle$$

Such a representation is called a *closure*. It is also the representation used by most Scheme and ML interpreters and compilers.

λ-calculus, briefly

The λ-calculus is a formal computation model (akin to Turing machines). It is built on the idea that everything is a function.

It has a minimal BNF grammar:

$$e ::= \mathtt{x} \;\mid\; (\lambda\ (\mathtt{x})\ e) \;\mid\; (e_0\, e_1)$$

Rather than give direct reduction/computation rules, we will translate expressions into a sub-system and give reduction rules for that system shortly.

The Ω-combinator is a one famous λ-expression which can be reduced indefinitely:

$((\lambda\ (\mathtt{x})\ (\mathtt{x}\,\mathtt{x}))\ (\lambda\ (\mathtt{y})\ (\mathtt{y}\,\mathtt{y})))$

From λ-calculus to ANF

$$e ::= \texttt{x} \mid (\lambda\ (\texttt{x})\ e) \mid (e_0\, e_1) \qquad \text{(lambda calculus)}$$

To make things easier for ourselves, we will bind the result of each intermediate computation to a name v.

$$
\begin{aligned}
P \ni \texttt{p} &::= \texttt{s} && \text{(programs)} \\
T \ni \texttt{t} &::= \texttt{x} \mid \texttt{v} \mid (\lambda\ (\texttt{x})\ \texttt{s}) && \text{(trivial expressions)} \\
C \ni \texttt{s} &::= \texttt{t} && \text{(serious expressions)} \\
&\quad\mid (\texttt{let}\ ((\texttt{v}\ \texttt{t}))\ \texttt{s}) \\
&\quad\mid (\texttt{t}_0\, \texttt{t}_1) \\
&\quad\mid (\texttt{let}\ ((\texttt{v}\ (\texttt{t}_0\, \texttt{t}_1)))\ \texttt{s})
\end{aligned}
$$

The grammar distinguishes *serious* expressions, (whose evaluation may diverge), from *trivial* expressions (whose evaluation will terminate).

Encoding control stacks as continuations

As a second step we will pass around our own control stack, encoded as a lambda term.

Hence every function will accept an additional parameter, *the continuation*.

Just as a plain control-stack tells us what to do next, our encoded stack (*the continuation*), tells us what to do next.

Encoding control stacks as continuations

As a second step we will pass around our own control stack, encoded as a lambda term.

Hence every function will accept an additional parameter, *the continuation*.

Just as a plain control-stack tells us what to do next, our encoded stack (*the continuation*), tells us what to do next.

Actually, we don't need to adhere to a stack-discipline, when we are implementing it ourselves (in the term).

Hence you can do funny stuff, like returning to the stack twice, not returning (i.e., jumping out of context), etc.

Example: continuation passing style

Consider an example:

```
(let ((f (λ (x) x)))
  ((f f) (λ (y) y)))
```

Example: continuation passing style

Consider an example:

```
(let ((f (λ (x) x)))
  ((f f) (λ (y) y)))
```

Sequentialized and with all intermediate computations named:

```
(let ((f (λ (x) x)))
  (let ((v (f f)))
    (v (λ (y) y))))
```

Example: continuation passing style

Consider an example:

```
(let ((f (λ (x) x)))
  ((f f) (λ (y) y)))
```

Sequentialized and with all intermediate computations named:

```
(let ((f (λ (x) x)))
  (let ((v (f f)))
    (v (λ (y) y))))
```

In continuation-passing style:

```
(λ (k0)  (let ((f (λ (x k) (k x))))
           (f f (λ (v)
             (v (λ (y k2) (k2 y)) k0)))))
```

Example: continuation passing style

Consider an example:

```
(let ((f (λ (x) x)))
  ((f f) (λ (y) y)))
```

Sequentialized and with all intermediate computations named:

```
(let ((f (λ (x) x)))
  (let ((v (f f)))
    (v (λ (y) y))))
```

In continuation-passing style:

```
(λ (k0)  ((λ  (f k1)
            (f f (λ (v)
              (v (λ (y k2) (k2 y)) k1))))
             (λ (x k) (k x)) k0))
```

CPS syntax

Formally, our grammar of CPS expressions is:

$$\begin{array}{rll} \mathit{CProg} \ni & \texttt{p} ::= (\lambda\ (\texttt{k})\ \texttt{e}) & \text{(CPS programs)} \\ \mathit{SExp} \ni & \texttt{e} ::= (\texttt{t}_0\,\texttt{t}_1\,\texttt{c}) \mid (\texttt{c}\ \texttt{t}) & \text{(serious expr)} \\ \mathit{TExp} \ni & \texttt{t} ::= \texttt{x} \mid \texttt{v} \mid (\lambda\ (\texttt{x}\ \texttt{k})\ \texttt{e}) & \text{(trivial expr)} \\ \mathit{CExp} \ni & \texttt{c} ::= (\lambda\ (\texttt{v})\ \texttt{e}) \mid \texttt{k} & \text{(continuation expr)} \end{array}$$

here expressed in Scheme syntax

Expressiveness

The language is still Turing-complete. We can express a CPS-version of the Ω-combinator $((\lambda\ (\mathtt{x})\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\ (\mathtt{y})\ (\mathtt{y}\ \mathtt{y})))$:

$$(\lambda\ (\mathtt{k}_0)\ ((\lambda\ (\mathtt{x}\ \mathtt{k}_1)\ (\mathtt{x}\ \mathtt{x}\ \mathtt{k}_1))\ (\lambda\ (\mathtt{y}\ \mathtt{k}_2)\ (\mathtt{y}\ \mathtt{y}\ \mathtt{k}_2))\ \mathtt{k}_0))$$

The CPS language represents the Church-side of the Church-Turing thesis.

One can thus Church-encode numbers:

$$c_0 = \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{z} \rightarrow (\lambda\ (\mathtt{s}\ \mathtt{k}_0)\ (\mathtt{k}_0\ (\lambda\ (\mathtt{z}\ \mathtt{k}_1)\ (\mathtt{k}_1\ \mathtt{z}))))$$

$$c_1 = \lambda \mathtt{s}.\lambda \mathtt{z}.(\mathtt{s}\ \mathtt{z}) \rightarrow (\lambda\ (\mathtt{s}\ \mathtt{k}_0)\ (\mathtt{k}_0\ (\lambda\ (\mathtt{z}\ \mathtt{k}_1)\ (\mathtt{s}\ \mathtt{z}\ (\lambda\ (\mathtt{v})\ (\mathtt{k}_1\ \mathtt{v})))$$

$$c_2 = \lambda \mathtt{s}.\lambda \mathtt{z}.(\mathtt{s}(\mathtt{s}\ \mathtt{z})) \rightarrow \ldots$$

CPS transforming ANF programs

Once programs are sequentialized and name all intermediate results, transforming into CPS is straightforward.

We formulate one transformation function for programs $\mathcal{C}$, for trivial terms $\mathcal{V}$, and for serious terms $\mathcal{F}$:

$$\mathcal{C} : P \rightarrow CProg$$
$$\mathcal{C}[\mathtt{p}] = \mathtt{(}\lambda\ \mathtt{(k_p)}\ \mathcal{F}_{\mathtt{k_p}}[\mathtt{p}]\mathtt{)}$$
where $\mathtt{k_p}$ is fresh

$$\mathcal{V} : T \rightarrow TExp$$
$$\mathcal{V}[\mathtt{x}] = \mathtt{x}$$
$$\mathcal{V}[\mathtt{(}\lambda\ \mathtt{(x)\ s)}] = \mathtt{(}\lambda\ \mathtt{(x\ k_s)}\ \mathcal{F}_{\mathtt{k_s}}[\mathtt{s}]\mathtt{)}$$
where $\mathtt{k_s}$ is fresh

$$\mathcal{F} : K \rightarrow C \rightarrow SExp$$
$$\mathcal{F}_{\mathtt{k}}[\mathtt{t}] = \mathtt{(k}\ \mathcal{V}[\mathtt{t}]\mathtt{)}$$
$$\mathcal{F}_{\mathtt{k}}[\mathtt{(let\ ((x\ t))\ s)}] = \mathtt{((}\lambda\ \mathtt{(x)}\ \mathcal{F}_{\mathtt{k}}[\mathtt{s}]\mathtt{)}\ \mathcal{V}[\mathtt{t}]\mathtt{)}$$
$$\mathcal{F}_{\mathtt{k}}[\mathtt{(t_0\ t_1)}] = \mathtt{(}\mathcal{V}[\mathtt{t_0}]\ \mathcal{V}[\mathtt{t_1}]\ \mathtt{k)}$$
$$\mathcal{F}_{\mathtt{k}}[\mathtt{(let\ ((x\ (t_0\ t_1)))\ s)}] = \mathtt{(}\mathcal{V}[\mathtt{t_0}]\ \mathcal{V}[\mathtt{t_1}]\ \mathtt{(}\lambda\ \mathtt{(x)}\ \mathcal{F}_{\mathtt{k}}[\mathtt{s}]\mathtt{))}$$

The CE abstract machine

Values and environments:

$$
\begin{aligned}
Val \ni \quad & w ::= [\texttt{(}\lambda\ \texttt{(x k) e)},\ r] \mid [\texttt{(}\lambda\ \texttt{(v) e)},\ r] \mid \texttt{stop} \\
Env \ni \quad & r\ ::= \bullet \mid r[\texttt{x} \mapsto w]
\end{aligned}
$$

The CE abstract machine

Values and environments:

$$
\begin{aligned}
\mathit{Val} \ni \quad & w ::= [(\lambda\ \texttt{(x k) e}), r] \mid [(\lambda\ \texttt{(v) e}), r] \mid \texttt{stop} \\
\mathit{Env} \ni \quad & r\ ::= \bullet \mid r[\texttt{x} \mapsto w]
\end{aligned}
$$

Two helper functions:

$$
\begin{aligned}
\mu_t : \mathit{TExp} \times \mathit{Env} & \rightharpoonup \mathit{Val} \\
\mu_t(\texttt{x}, r) &= r(\texttt{x}) \\
\mu_t(\texttt{v}, r) &= r(\texttt{v}) \\
\mu_t((\lambda\ \texttt{(x k) e}), r) &= [(\lambda\ \texttt{(x k) e}), r]
\end{aligned}
$$

$$
\begin{aligned}
\mu_c : \mathit{CExp} \times \mathit{Env} & \rightharpoonup \mathit{Val} \\
\mu_c(\texttt{k}, r) &= r(\texttt{k}) \\
\mu_c((\lambda\ \texttt{(v) e}), r) &= [(\lambda\ \texttt{(v) e}), r]
\end{aligned}
$$

The CE abstract machine

Values and environments:

$$Val \ni \quad w ::= [\texttt{(}\lambda\ \texttt{(x k) e)},\ r] \mid [\texttt{(}\lambda\ \texttt{(v) e)},\ r] \mid \texttt{stop}$$
$$Env \ni \quad r ::= \bullet \mid r[\texttt{x} \mapsto w]$$

Two helper functions:

$$\mu_t : TExp \times Env \rightharpoonup Val$$
$$\mu_t(\texttt{x}, r) = r(\texttt{x})$$
$$\mu_t(\texttt{v}, r) = r(\texttt{v})$$
$$\mu_t(\texttt{(}\lambda\ \texttt{(x k) e)}, r) = [\texttt{(}\lambda\ \texttt{(x k) e)},\ r]$$

$$\mu_c : CExp \times Env \rightharpoonup Val$$
$$\mu_c(\texttt{k}, r) = r(\texttt{k})$$
$$\mu_c(\texttt{(}\lambda\ \texttt{(v) e)}, r) = [\texttt{(}\lambda\ \texttt{(v) e)},\ r]$$

Transition relation (over $SExp \times Env$):

$$\langle \texttt{(t}_0\ \texttt{t}_1\ \texttt{c)},\ r\rangle \longrightarrow \langle \texttt{e},\ r'[\texttt{x} \mapsto w][\texttt{k} \mapsto w_\texttt{c}]\rangle \quad \text{if } [\texttt{(}\lambda\ \texttt{(x k) e)},\ r'] = \mu_t(\texttt{t}_0, r),\ w = \mu_t(\texttt{t}_1, r),\ w_\texttt{c} = \mu_c(\texttt{c}, r)$$

$$\langle \texttt{(c t)},\ r\rangle \longrightarrow \langle \texttt{e},\ r'[\texttt{v} \mapsto w]\rangle \quad \text{if } [\texttt{(}\lambda\ \texttt{(v) e)},\ r'] = \mu_c(\texttt{c}, r),\ w = \mu_t(\texttt{t}, r)$$

Initial state:

$\langle \texttt{e},\ \bullet[\texttt{k} \mapsto [\texttt{(}\lambda\ \texttt{(v}_r\texttt{) (k}_r\ \texttt{v}_r\texttt{))},\ \bullet[\texttt{k}_r \mapsto \texttt{stop}]]]\rangle$ for program $\texttt{(}\lambda\ \texttt{(k) e)}$

Exercise

Trace the first four steps of the CE-machine on the Ω-combinator in CPS:

$$(\lambda\ (\mathtt{k}_0)\ ((\lambda\ (\mathtt{x}\ \mathtt{k}_1)\ (\mathtt{x}\ \mathtt{x}\ \mathtt{k}_1))\ (\lambda\ (\mathtt{y}\ \mathtt{k}_2)\ (\mathtt{y}\ \mathtt{y}\ \mathtt{k}_2))\ \mathtt{k}_0))$$

To CPS transform or not to CPS transform

Note: we don't need to CPS transform terms to give an abstract machine semantics.

Flanagan-al:PLDI93 (optional reading) provides alternative abstract machines for non-CPS-transformed terms.

Ager-al:PPDP03 (also optional reading) suggests a systematic approach to construct even more by yourselves.

IMP semantics

IMP programs

We'll study a simple imperative language IMP, composed of statements, arithmetic expressions, and boolean expressions:

$$
\begin{aligned}
s \ni \mathit{Stmt} ::= {} & \mathtt{x} = e \\
\mid {} & \mathtt{skip} \\
\mid {} & \mathtt{if}\ \mathit{test}\ \mathtt{then}\ s\ \mathtt{else}\ s \\
\mid {} & \mathtt{while}\ \mathit{test}\ \mathtt{do}\ s \\
\mid {} & s\,;\,s
\end{aligned}
$$

$$
\begin{aligned}
e \ni \mathit{AExp} ::= {} & \mathtt{n} \\
\mid {} & ? \\
\mid {} & \mathtt{x} \\
\mid {} & e\ op\ e
\end{aligned}
$$

where $op \in \{+, -, *, \ldots\}$

$$
\begin{aligned}
\mathit{test} \ni \mathit{BExp} ::= {} & e\ \mathit{comp}\ e \\
\mid {} & \mathit{test}\ \mathtt{and}\ \mathit{test} \\
\mid {} & \mathit{test}\ \mathtt{or}\ \mathit{test}
\end{aligned}
$$

where $\mathit{comp} \in \{=, <>, <, \ldots\}$

Note: because of $?$, programs are non-deterministic.

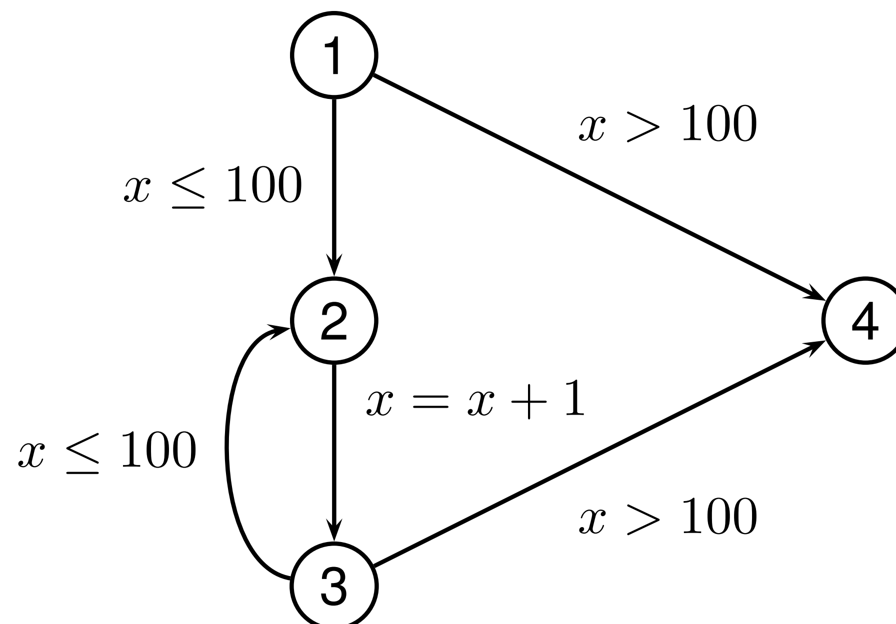
Imperative programs as flow graphs

Rather than giving a direct semantics, we will represent simple imperative programs using their *flow graph* (or *flow chart*).

We associate program actions (tests, assignments, etc.) to the edges of the graph (instead of associating them to the nodes of the graph).

Example:

```
while x<=100 {
  x = x + 1;
}
```



Flow graphs, formally

Formally, a program graph is a quadruple $\langle V,\ v_{entry},\ v_{exit},\ E\rangle$, where

- V is a finite set of vertices
- $E \subseteq V \times V$ is a finite set of edges
- $v_{entry} \in V$ is a distinct entry vertex (in-degree 0)
- $v_{exit} \in V$ is a distinct exit vertex (out-degree 0)

Every vertex lies on a path from v_{entry} to v_{exit}.

Imperative programs as flow graphs, formally

Instructions are divided into assignments and tests:

$$
\begin{aligned}
I ::= {} & \mathtt{x} = e \\
\mid {} & \mathtt{assert}\ test
\end{aligned}
$$

A program is a triple $\langle G, U, L \rangle$, where

- the program graph G
- the universe U of variables, ($\mathtt{x}, \mathtt{y} \in U$)
- the labelling function $L \in (E \to I)$ associating an instruction to each edge

Semantics of arithmetic expressions and tests

A store remembers the program state: $\rho \ni Store = U \to \mathbb{Z}$

$$\mathcal{A} : AExp \to Store \to \wp(\mathbb{Z})$$

$$\mathcal{A}\,[\![\mathtt{n}]\!]\,\rho = \{n\}$$

$$\mathcal{A}\,[\![?]\!]\,\rho = \mathbb{Z}$$

$$\mathcal{A}\,[\![\mathtt{x}]\!]\,\rho = \{\rho(\mathtt{x})\}$$

$$\mathcal{A}\,[\![e \;\; op \;\; e']\!]\,\rho = \{n \;\; op \;\; n' \mid n \in \mathcal{A}\,[\![e]\!]\,\rho, n' \in \mathcal{A}\,[\![e']\!]\,\rho\} \quad \text{where} \quad op \in \{+, -, *, \ldots\}$$

Note: by computing over $\mathbb{Z}$ we are ignoring overflow.

Semantics of arithmetic expressions and tests

A store remembers the program state: $\rho \ni Store = U \to \mathbb{Z}$

$$\mathcal{A} : AExp \to Store \to \wp(\mathbb{Z})$$

$$\mathcal{A}\llbracket \texttt{n} \rrbracket \rho = \{n\}$$

$$\mathcal{A}\llbracket ? \rrbracket \rho = \mathbb{Z}$$

$$\mathcal{A}\llbracket \texttt{x} \rrbracket \rho = \{\rho(\texttt{x})\}$$

$$\mathcal{A}\llbracket e \;\; op \;\; e' \rrbracket \rho = \{n \;\; op \;\; n' \mid n \in \mathcal{A}\llbracket e \rrbracket \rho, n' \in \mathcal{A}\llbracket e' \rrbracket \rho\} \qquad \text{where} \quad op \in \{+, -, *, \dots\}$$

Note: by computing over $\mathbb{Z}$ we are ignoring overflow.

$$\mathcal{B} : BExp \to Store \to \wp(\mathbb{B}) \qquad \text{where} \;\; \mathbb{B} = \{true, false\}$$

$$\mathcal{B}\llbracket e \;\; comp \;\; e' \rrbracket \rho = \{true \mid n \in \mathcal{A}\llbracket e \rrbracket \rho \;\wedge\; n' \in \mathcal{A}\llbracket e' \rrbracket \rho \;\wedge\; n \;\; comp \;\; n'\}$$
$$\bigcup \{false \mid n \in \mathcal{A}\llbracket e \rrbracket \rho \;\wedge\; n' \in \mathcal{A}\llbracket e' \rrbracket \rho \;\wedge\; \neg(n \;\; comp \;\; n')\}$$

$$\mathcal{B}\llbracket test \; \texttt{and} \; test' \rrbracket \rho = \{b \;\wedge\; b' \mid b \in \mathcal{B}\llbracket test \rrbracket \rho \;\wedge\; b' \in \mathcal{B}\llbracket test' \rrbracket \rho\}$$

$$\mathcal{B}\llbracket test \; \texttt{or} \; test' \rrbracket \rho = \{b \;\vee\; b' \mid b \in \mathcal{B}\llbracket test \rrbracket \rho \;\wedge\; b' \in \mathcal{B}\llbracket test' \rrbracket \rho\}$$

IMP program execution as a transition system

States are pairs:

$$State = V \times Store$$

There is one case per instruction:

$$\langle v,\ \rho\rangle \to \langle v',\ \rho[\mathtt{x} \mapsto n]\rangle \qquad \text{if} \quad \begin{array}{c} \langle v,\ v'\rangle \in E\ \wedge \\ L(\langle v,\ v'\rangle) = (\mathtt{x} = e)\ \wedge \\ n \in \mathcal{A}\,[\![e]\!]\,\rho \end{array}$$

$$\langle v,\ \rho\rangle \to \langle v',\ \rho\rangle \qquad \text{if} \quad \begin{array}{c} \langle v,\ v'\rangle \in E\ \wedge \\ L(\langle v,\ v'\rangle) = (\mathtt{assert}\ test)\ \wedge \\ true \in \mathcal{B}\,[\![test]\!]\,\rho \end{array}$$

Initial state: $\langle v_{entry},\ \rho\rangle$ (for initial store ρ)

IMP program execution as a transition system

States are pairs:

$$State = V \times Store$$

There is one case per instruction:

$$\langle v,\, \rho \rangle \to \langle v',\, \rho[\mathtt{x} \mapsto n] \rangle \qquad \text{if} \quad \begin{array}{c} \langle v,\, v' \rangle \in E \;\wedge \\ L(\langle v,\, v' \rangle) = (\mathtt{x} = e) \;\wedge \\ n \in \mathcal{A}\,[\![e]\!]\,\rho \end{array}$$

$$\langle v,\, \rho \rangle \to \langle v',\, \rho \rangle \qquad \text{if} \quad \begin{array}{c} \langle v,\, v' \rangle \in E \;\wedge \\ L(\langle v,\, v' \rangle) = (\mathtt{assert}\; test) \;\wedge \\ true \in \mathcal{B}\,[\![test]\!]\,\rho \end{array}$$

Initial states: $\{\langle v_{entry},\, \rho \rangle \mid \rho \in Store\}$ (for initial store ρ)

Bytecode semantics

Scope, mutation, and semantics

The CPS semantics illustrates how to model binding and lexical scope, namely with environments.

The flow-graph semantics illustrates how to model mutation, namely with a global store.

The bytecode semantics can express both — in addition to heap-allocated objects. It is hence a bit more complex.

A JVM-like instruction set

$$
\begin{array}{rlll}
\mathit{Inst} ::= & \texttt{nop} & \mid \texttt{numop}\ op & \mid \texttt{new}\ cl \\
\mid & \texttt{push}\ c & \mid \texttt{load}\ i & \mid \texttt{putfield}\ f \\
\mid & \texttt{pop} & \mid \texttt{store}\ i & \mid \texttt{getfield}\ f \\
\mid & \texttt{dup} & \mid \texttt{ifeq}\ pc & \mid \texttt{invokevirtual}\ M \\
\mid & \texttt{swap} & \mid \texttt{goto}\ pc & \mid \texttt{return}
\end{array}
$$

where $op \in \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}, \texttt{rem}, \texttt{and}, \texttt{or}, \ldots\}$

Numeric operations are collected in one bytecode.

$$
\begin{aligned}
pc \ni \mathit{Address} &= \mathbb{N} \\
m \ni \mathit{Method} &= \mathit{MethodId} \times (\mathit{Address} \to \mathit{Inst}) \\
\mathit{Field} &= \mathit{FieldName} \\
c \ni \mathit{Class} &= \mathit{ClassName} \times \mathit{Class}_\perp \times \wp(\mathit{Field}) \times \wp(\mathit{Method}) \\
P \ni \mathit{Program} &= \wp(\mathit{Class})
\end{aligned}
$$

Virtual machine domains

$$
\begin{aligned}
loc \ni Locations & \quad \text{(some countable number of locations)} \\
v \ni Value &= n \mid loc \mid \texttt{null} \\
s \ni OperandStack &= Value^* \\
l \ni LocalVar &= [Value_\bot] \\
Frame &= Method \times Address \times LocalVar \times OperandStack \\
sf \ni CallStack &= Frame^* \\
o \ni Object &= Class \times (FieldName \rightharpoonup Value) \\
h \ni Heap &= Locations \to Object_\bot \\
State &= Heap \times CallStack
\end{aligned}
$$

We now define a number of shorthands and helper functions:

$$className(c) = \pi_1(c) \qquad methodName(m) = \pi_1(m) \qquad instAt_P(m, pc) = \pi_2(m)(pc)$$

$$methods(c) = \pi_4(c) \qquad class(o) = \pi_1(o) \qquad fieldValue(o, f) = \pi_2(o)(f)$$

$$newObject(h, c) = \langle h[loc \mapsto \langle c, \bullet \rangle], loc \rangle \text{ where } loc \notin Dom(h)$$

$$lookup(M, c) = \begin{cases} m & \text{if } m \in methods(c) \wedge methodName(m) = M \\ lookup(M, \pi_2(c)) & \text{if } \pi_2(c) \neq \bot \wedge \langle M, \pi_2(c) \rangle \in Dom(lookup) \end{cases}$$

Byte code execution (1/3)

$$\frac{instAt_P(m, pc) = \texttt{nop}}{\langle h,\ (m,\ pc,\ l,\ s) :: sf\rangle \to \langle h,\ (m,\ pc+1,\ l,\ s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{push}\ c}{\langle h,\ (m,\ pc,\ l,\ s) :: sf\rangle \to \langle h,\ (m,\ pc+1,\ l,\ c :: s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{pop}}{\langle h,\ (m,\ pc,\ l,\ v :: s) :: sf\rangle \to \langle h,\ (m,\ pc+1,\ l,\ s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{dup}}{\langle h,\ (m,\ pc,\ l,\ v :: s) :: sf\rangle \to \langle h,\ (m,\ pc+1,\ l,\ v :: v :: s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{swap}}{\langle h,\ (m,\ pc,\ l,\ v_1 :: v_2 :: s) :: sf\rangle \to \langle h,\ (m,\ pc+1,\ l,\ v_2 :: v_1 :: s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{numop}\ op}{\langle h,\ (m,\ pc,\ l,\ n_1 :: n_2 :: s) :: sf\rangle \to \langle h,\ (m,\ pc+1,\ l,\ [\![op]\!](n_1,\ n_2) :: s) :: sf\rangle}$$

Byte code execution (2/3)

$$\frac{instAt_P(m, pc) = \texttt{load}\ i}{\langle h,\ (m,\ pc,\ l,\ s) :: sf\rangle \rightarrow \langle h,\ (m,\ pc+1,\ l,\ l(i) :: s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{store}\ i}{\langle h,\ (m,\ pc,\ l,\ v :: s) :: sf\rangle \rightarrow \langle h,\ (m,\ pc+1,\ l[i \mapsto v],\ s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{ifeq}\ pc' \qquad n = 0}{\langle h,\ (m,\ pc,\ l,\ n :: s) :: sf\rangle \rightarrow \langle h,\ (m,\ pc',\ l,\ s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{ifeq}\ pc' \qquad n \neq 0}{\langle h,\ (m,\ pc,\ l,\ n :: s) :: sf\rangle \rightarrow \langle h,\ (m,\ pc+1,\ l,\ s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{goto}\ pc'}{\langle h,\ (m,\ pc,\ l,\ s) :: sf\rangle \rightarrow \langle h,\ (m,\ pc',\ l,\ s) :: sf\rangle}$$

$$\frac{\begin{array}{c} instAt_P(m, pc) = \texttt{new}\ cl \\ \exists c \in classes(P) : className(c) = cl \qquad \langle h',\ loc\rangle = newObject(h, c) \end{array}}{\langle h,\ (m,\ pc,\ l,\ s) :: sf\rangle \rightarrow \langle h',\ (m,\ pc+1,\ l,\ loc :: s) :: sf\rangle}$$

Byte code execution (3/3)

$$\frac{instAt_P(m, pc) = \texttt{putfield}\ f \qquad h(loc) = o \qquad o' = \langle class(o),\ \pi_2(o)[f \mapsto v]\rangle}{\langle h,\ (m,\ pc,\ l,\ v :: loc :: s) :: sf\rangle \to \langle h[loc \mapsto o'],\ (m,\ pc+1,\ l,\ s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{getfield}\ f \qquad h(loc) = o}{\langle h,\ (m,\ pc,\ l,\ loc :: s) :: sf\rangle \to \langle h,\ (m,\ pc+1,\ l,\ fieldValue(o, f) :: s) :: sf\rangle}$$

$$\frac{\begin{array}{c} instAt_P(m, pc) = \texttt{invokevirtual}\ M \\ h(loc) = o \qquad m' = lookup(M, class(o)) \end{array}}{\langle h,\ (m,\ pc,\ l,\ loc :: \vec{v} :: s) :: sf\rangle \to \langle h,\ (m',\ 1,\ loc \cdot \vec{v},\ \epsilon) :: (m,\ pc,\ l,\ s) :: sf\rangle}$$

$$\frac{instAt_P(m, pc) = \texttt{return}}{\langle h,\ (m,\ pc,\ l,\ v :: s) :: (m',\ pc',\ l',\ s') :: sf\rangle \to \langle h,\ (m',\ pc'+1,\ l',\ v :: s') :: sf\rangle}$$

Initial state:

$$\langle \bullet,\ (lookup(\texttt{main}, c),\ 1,\ \epsilon,\ \epsilon) :: \epsilon\rangle$$

for program P and class c.

Collecting semantics, revisited

Collecting semantics, revisited (1/3)

We formulate the collecting semantics in terms of sets because sets describe properties, e.g.,

- the set $\{1, 3, 5, \ldots\}$ describes the property odd
- the set $\{2, 4, 6, \ldots\}$ describes the property $even$
- the singleton set $\{42\}$ describes a constant property
- the set $\{4, 5, 6, 7, 8, 9, 10\}$ describes an interval property $[4; 10]$
- $\ldots$

In this sense, the collecting semantics is the strongest property expressed as a (generally uncomputable) fixed point.

Collecting semantics, revisited (2/3)

The collecting semantics can be viewed as a logic.

In our case the reachable states collecting semantics over $\langle \wp(S); \subseteq, \emptyset, S, \cup, \cap \rangle$ can be understood as follows.

- $\subseteq$ is implication
- $\emptyset$ is false
- S is true
- $\cup$ is disjunction
- $\cap$ is conjunction

A (post-)fixed point Σ' of F then satisfies $F(\Sigma') \subseteq \Sigma'$ (read: "$F(\Sigma') \implies \Sigma'$"), which means that Σ' is an *invariant* for the reachable states.

Collecting semantics, revisited (3/3)

In more detail: A post-fixed point Σ' of $F(\Sigma) = S_i \cup \{s' \mid \exists s \in \Sigma : s \to s'\}$ satisfies:

- $S_i \subseteq \Sigma'$ $\sim$ "The initial state satisfies Σ'"
- $\{s' \mid \exists s \in \Sigma' : s \to s'\} \subseteq \Sigma'$
  $\sim$ "Σ' is preserved across transitions"

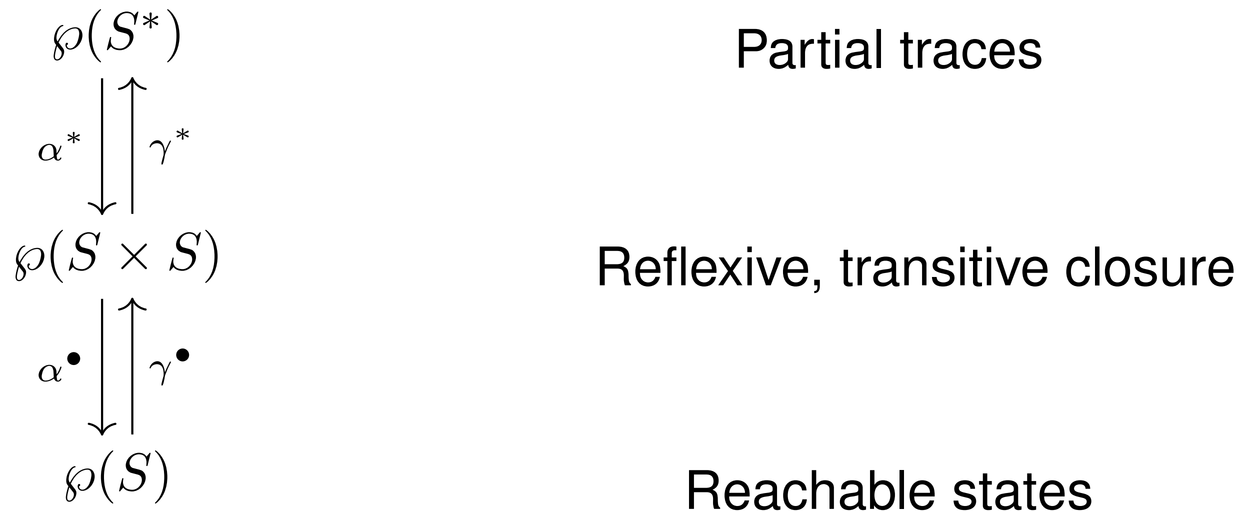
Thus Σ' is an invariant.

A fixed point computation describes the iterative search for an invariant in this logic.

Note: any post-fixed point of F is a valid invariant (but some are more interesting that others. . .)

Stronger properties, stronger collecting semantics

There is a hierarchy of increasingly powerful collecting semantics:



Stronger properties, stronger collecting semantics

There is a hierarchy of increasingly powerful collecting semantics:

$$\begin{array}{ll} \wp(S^*) & \lambda X.\, \{s \mid s \in S\} \cup \{\sigma s s' \mid \sigma s \in X \ \wedge \ s \to s'\} \\ \alpha^* \downarrow \uparrow \gamma^* & \\ \wp(S \times S) & \lambda Y.\, \{\langle s,\, s\rangle \mid s \in S\} \cup \{\langle s,\, s''\rangle \mid \exists s' : \langle s,\, s'\rangle \in Y \ \wedge \ s' \to s''\} \\ \alpha^\bullet \downarrow \uparrow \gamma^\bullet & \\ \wp(S) & \lambda Z.\, S_i \cup \{s' \mid \exists s \in Z : s \to s'\} \end{array}$$

Each can be expressed as a least fixed point

Fun with the three counter machine

Recall Plotkin's three counter machine (1/2)

There are 3 variables (or registers):

$$var \in Var = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \qquad \text{(variables)}$$

$$\begin{aligned} Inst ::= {} & \mathtt{inc}\ var \\ \mid {} & \mathtt{dec}\ var \\ \mid {} & \mathtt{zero}\ var\ pc\ \mathtt{else}\ pc' \\ \mid {} & \mathtt{stop} \end{aligned} \qquad \text{(instructions)}$$

$$P = Inst^* \qquad \text{(programs)}$$

$$pc \in PC = \mathbb{N} \qquad \text{(program counter)}$$

$$States = PC \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \qquad \text{(states)}$$

Initial states: $\{\langle 1,\ i,\ 0,\ 0\rangle \mid i \in \mathbb{N}_0\}$ (for program P with input i)

Final states: $\{\langle pc,\ 0,\ yv,\ 0\rangle \mid pc \in PC\ \wedge\ yv \in \mathbb{N}_0\ \wedge\ P_{pc} = \mathtt{stop}\}$
(with yv being the result)

Recall Plotkin's three counter machine (2/2)

Transition relation:

$$\begin{array}{rcll}
\langle pc,\ xv,\ yv,\ zv\rangle & \longrightarrow & \langle pc+1,\ xv+1,\ yv,\ zv\rangle & \text{if } P_{pc} = \texttt{inc x} \\
- & \longrightarrow & \langle pc+1,\ xv,\ yv+1,\ zv\rangle & \text{if } P_{pc} = \texttt{inc y} \\
- & \longrightarrow & \langle pc+1,\ xv,\ yv,\ zv+1\rangle & \text{if } P_{pc} = \texttt{inc z} \\
\langle pc,\ xv,\ yv,\ zv\rangle & \longrightarrow & \langle pc+1,\ xv-1,\ yv,\ zv\rangle & \text{if } P_{pc} = \texttt{dec x} \ \wedge\ xv > 0 \\
- & \longrightarrow & \langle pc+1,\ xv,\ yv-1,\ zv\rangle & \text{if } P_{pc} = \texttt{dec y} \ \wedge\ yv > 0 \\
- & \longrightarrow & \langle pc+1,\ xv,\ yv,\ zv-1\rangle & \text{if } P_{pc} = \texttt{dec z} \ \wedge\ zv > 0 \\
\langle pc,\ xv,\ yv,\ zv\rangle & \longrightarrow & \langle pc',\ xv,\ yv,\ zv\rangle & \text{if } P_{pc} = \texttt{zero x}\ pc'\ \texttt{else}\ pc'' \ \wedge\ xv = 0 \\
- & \longrightarrow & \langle pc'',\ xv,\ yv,\ zv\rangle & \text{if } P_{pc} = \texttt{zero x}\ pc'\ \texttt{else}\ pc'' \ \wedge\ xv \neq 0 \\
\langle pc,\ xv,\ yv,\ zv\rangle & \longrightarrow & \langle pc',\ xv,\ yv,\ zv\rangle & \text{if } P_{pc} = \texttt{zero y}\ pc'\ \texttt{else}\ pc'' \ \wedge\ yv = 0 \\
- & \longrightarrow & \langle pc'',\ xv,\ yv,\ zv\rangle & \text{if } P_{pc} = \texttt{zero y}\ pc'\ \texttt{else}\ pc'' \ \wedge\ yv \neq 0 \\
\langle pc,\ xv,\ yv,\ zv\rangle & \longrightarrow & \langle pc',\ xv,\ yv,\ zv\rangle & \text{if } P_{pc} = \texttt{zero z}\ pc'\ \texttt{else}\ pc'' \ \wedge\ zv = 0 \\
- & \longrightarrow & \langle pc'',\ xv,\ yv,\ zv\rangle & \text{if } P_{pc} = \texttt{zero z}\ pc'\ \texttt{else}\ pc'' \ \wedge\ zv \neq 0
\end{array}$$

Plotkin's three counter machine in ASCII. . .

Transition relation:

```
<pc, xv, yv, zv> --> <pc+1, xv+1, yv, zv>                   if P_pc = inc x
        -        --> <pc+1, xv, yv+1, zv>                   if P_pc = inc y
        -        --> <pc+1, xv, yv, zv+1>                   if P_pc = inc z

<pc, xv, yv, zv> --> <pc+1, xv-1, yv, zv>           if P_pc = dec x /\ xv>0
        -        --> <pc+1, xv, yv-1, zv>           if P_pc = dec y /\ yv>0
        -        --> <pc+1, xv, yv, zv-1>           if P_pc = dec z /\ zv>0

<pc, xv, yv, zv> --> <pc', xv, yv, zv>      if P_pc = zero x pc' else pc''
                                               /\ xv=0
        -        --> <pc'', xv, yv, zv>     if P_pc = zero x pc' else pc''
                                               /\ xv<>0

<pc, xv, yv, zv> --> <pc', xv, yv, zv>      if P_pc = zero y pc' else pc''
                                               /\ yv=0
        -        --> <pc'', xv, yv, zv>     if P_pc = zero y pc' else pc''
                                               /\ yv<>0

<pc, xv, yv, zv> --> <pc', xv, yv, zv>      if P_pc = zero z pc' else pc''
                                               /\ zv=0
        -        --> <pc'', xv, yv, zv>     if P_pc = zero z pc' else pc''
                                               /\ zv<>0
```

Implementation of the three counter machine

Quick tour of implementation:

- AST
- Lexer
- Parser
- Wellformedness (checks out of bounds)
- Interpreter

Each of the above reside in their own module (and file).

To build from scratch run: `make` (requires OCaml)

Download: `https://github.com/jmid/3CounterMach`

An epigram from Perlis

Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

— Alan Perlis

Formulating the collecting semantics

Recall the reachable states collecting semantics:

$$F(\Sigma) = S_i \cup \{\sigma \mid \exists \sigma' \in \Sigma : \sigma' \to \sigma\}$$

Let's write the specialized version. . .

Abstracting the collecting semantics

We abstract the collecting semantics to a set valued function using the well-known Galois connection:

$$\langle \wp(A \times B); \subseteq, \emptyset, A \times B, \cup, \cap \rangle \xleftarrow[\alpha]{\gamma} \langle A \to \wp(B); \dot{\subseteq}, \lambda x.\emptyset, \lambda x.B, \dot{\cup}, \dot{\cap} \rangle$$

where
$$\alpha(R) = \lambda a.\{b \mid (a, b) \in R\}$$
$$\gamma(F) = \{(a, b) \mid b \in F(a)\}$$

Note: in our case $A = PC$ and $B = \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0$.

We'll use the "alpha-gamma" composition approach. . .

Result

```
F#(S#)  =  Ø. [1 -> { <i,0,0> | i in N_0 }]

  U.
            U.         Ø. [pc+1 -> { <xv+1, yv, zv> }]
   { <xv, yv, zv> } C S#(pc)
      P_pc = inc x                 (...and for y and z)
  U.
            U.         Ø. [pc+1 -> { <xv-1, yv, zv> }]
   { <xv, yv, zv> } C S#(pc)
      P_pc = dec x
          xv>0                     (...and for y and z)
  U.
            U.         Ø. [pc' -> { <xv, yv, zv> }]
   { <xv, yv, zv> } C S#(pc)
  P_pc = zero x pc' else pc''
           xv=0                    (...and for y and z)
  U.
            U.         Ø. [pc'' -> { <xv, yv, zv> }]
   { <xv, yv, zv> } C S#(pc)
  P_pc = zero x pc' else pc''
          xv<>0                    (...and for y and z)
```

What happened?

We systematically massaged the transition function of the collecting semantics

$$F : \wp(PC \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0) \to \wp(PC \times \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0)$$

into a transition function over a related domain

$$F\# : (PC \to \wp(\mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0)) \to (PC \to \wp(\mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0))$$

by surfing on the Galois connections.

Note: a least fixed point of the resulting function is still not computable (after all, the domains are isomorphic), so we are not quite there yet. . .

Cliffhanger...

To be continued...

Summary

Summary

We've seen four different abstract machine semantics:

- Plotkin's three counter machine
- the CE machine for CPS programs
- a flow-chart semantics for IMP programs
- a JVM-like semantics for bytecodes

Finally we took

- another look at collecting semantics and
- the first step towards analysing Plotkin's 3 counter machine