# Semantics

Jan Midtgaard

Week 2, Abstract Interpretation

Aarhus University, Q4 - 2012

# Last time

- 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):

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

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

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

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

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

Initial 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):

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

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

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

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

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

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)

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

Transition relation:

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

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 evaluating:

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

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.

# From $\lambda$-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 $\texttt{v}$.

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

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.

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:

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

here expressed in Scheme syntax

# Expressiveness

The language is Turing-complete, in that it is sufficient to express a CPS-version of the $\Omega$-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:

$$\begin{aligned}
c_0 &= \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{z} && \rightarrow (\lambda\ (\mathtt{s}\ \mathtt{k}_0)\ (\mathtt{k}_0\ (\lambda\ (\mathtt{z}\ \mathtt{k}_1)\ (\mathtt{k}_1\ \mathtt{z}))))\\
c_1 &= \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}\ \mathtt{z} && \rightarrow (\lambda\ (\mathtt{s}\ \mathtt{k}_0)\ (\mathtt{k}_0\ (\lambda\ (\mathtt{z}\ \mathtt{k}_1)\ (\mathtt{s}\ \mathtt{z}\ (\lambda\ (\mathtt{v})\ (\mathtt{k}_1\ \mathtt{v}))))\\
c_2 &= \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}(\mathtt{s}\ \mathtt{z}) && \rightarrow \ldots
\end{aligned}$$

# 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 $\mathcal{F}$:

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

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

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

# The CE abstract machine

Values and environments:

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

# The CE abstract machine

Values and environments:

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

Two helper functions:

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

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

# The CE abstract machine

Values and environments:

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

Two helper functions:

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

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

Transition relation (over $\mathit{SExp} \times \mathit{Env}$):

$$\langle \texttt{(t}_0\,\texttt{t}_1\,\texttt{c)},\ r\rangle \longrightarrow \langle \texttt{e},\ r'[\texttt{x} \mapsto w][\texttt{k} \mapsto w_{\texttt{c}}]\rangle$$

if $[\texttt{(}\lambda\ \texttt{(x k) e)},\ r'] = \mu_t(\texttt{t}_0, r)$

$$\begin{aligned} w &= \mu_t(\texttt{t}_1, r) \\ w_{\texttt{c}} &= \mu_c(\texttt{c}, r) \end{aligned}$$

$$\langle \texttt{(c t)},\ r\rangle \longrightarrow \langle \texttt{e},\ r'[\texttt{v} \mapsto w]\rangle$$

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 $\Omega$-combinator in CPS:

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

# 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.

From Olivier's TFP course some of you know how 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:

$$
\begin{aligned}
s \ni \mathit{Stmt} ::= {} & \mathtt{x} = e \\
\mid {} & \mathtt{skip} \\
\mid {} & \mathtt{if}\ \mathit{test}\ \mathtt{then}\ s\ \mathtt{else}\ s \\
\mid {} & \mathtt{while}\ \mathit{test}\ \mathtt{do}\ s \\
\mid {} & s\,;\,s
\end{aligned}
\qquad
\begin{aligned}
e \ni \mathit{AExp} ::= {} & \mathtt{n} \\
\mid {} & ? \\
\mid {} & \mathtt{x} \\
\mid {} & e\ \mathit{op}\ e \\
& \text{where} \quad \mathit{op} \in \{+, -, *, \dots\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{test} \ni \mathit{BExp} ::= {} & e\ \mathit{comp}\ e \quad \text{where} \quad \mathit{comp} \in \{=, <>, <, \dots\} \\
\mid {} & \mathit{test}\ \mathtt{and}\ \mathit{test} \\
\mid {} & \mathit{test}\ \mathtt{or}\ \mathit{test}
\end{aligned}
$$

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

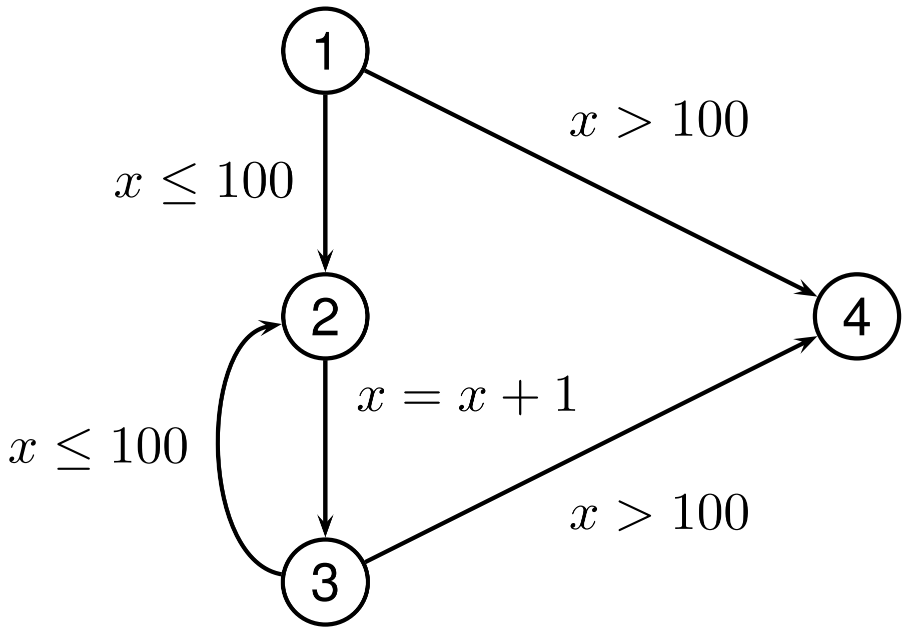
# Imperative programs as flow graphs

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: $\rho \ni \mathit{Store} = U \to \mathbb{Z}$

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

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

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

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

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

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

# Semantics of arithmetic expressions and tests

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

$$
\begin{aligned}
\mathcal{A} : \mathit{AExp} \to \mathit{Store} \to \wp(\mathbb{Z}) \\
\mathcal{A}\,[\![\texttt{n}]\!]\,\rho &= \{n\} \\
\mathcal{A}\,[\![?]\!]\,\rho &= \mathbb{Z} \\
\mathcal{A}\,[\![\texttt{x}]\!]\,\rho &= \{\rho(\texttt{x})\} \\
\mathcal{A}\,[\![e\ op\ e']\!]\,\rho &= \{n\ op\ n' \mid n \in \mathcal{A}\,[\![e]\!]\,\rho, n' \in \mathcal{A}\,[\![e']\!]\,\rho\} \qquad \text{where} \quad op \in \{+, -, *, \dots\}
\end{aligned}
$$

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

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

$$
\begin{aligned}
\mathcal{B}\,[\![e\ comp\ e']\!]\,\rho &= \begin{array}{l} \{\mathit{true} \mid n \in \mathcal{A}\,[\![e]\!]\,\rho \ \wedge\ n' \in \mathcal{A}\,[\![e]\!]\,\rho \ \wedge\ n\ comp\ n'\} \\ \bigcup \{\mathit{false} \mid n \in \mathcal{A}\,[\![e]\!]\,\rho \ \wedge\ n' \in \mathcal{A}\,[\![e]\!]\,\rho \ \wedge\ \neg(n\ comp\ n')\} \end{array} \\
\mathcal{B}\,[\![\mathit{test}\ \texttt{and}\ \mathit{test}']\!]\,\rho &= \{b \ \wedge\ b' \mid b \in \mathcal{B}\,[\![\mathit{test}]\!]\,\rho \ \wedge\ b' \in \mathcal{B}\,[\![\mathit{test}']\!]\,\rho\} \\
\mathcal{B}\,[\![\mathit{test}\ \texttt{or}\ \mathit{test}']\!]\,\rho &= \{b \ \vee\ b' \mid b \in \mathcal{B}\,[\![\mathit{test}]\!]\,\rho \ \wedge\ b' \in \mathcal{B}\,[\![\mathit{test}']\!]\,\rho\}
\end{aligned}
$$

# 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 $\rho$)

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

# Bytecode semantics

# Scope, mutation, and semantics

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

The flow-graph semantics tells us 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

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

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

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

We now define a number of shorthands and helper functions:

$$
\begin{aligned}
\mathit{className}(c) &= \pi_1(c) & \mathit{methodName}(m) &= \pi_1(m) & \mathit{instAt}_P(m, pc) &= \pi_2(m)(pc) \\
\mathit{methods}(c) &= \pi_4(c) & \mathit{class}(o) &= \pi_1(o) & \mathit{fieldValue}(o, f) &= \pi_2(o)(f)
\end{aligned}
$$

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

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

# Byte code execution (1/3)

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

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

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

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

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

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

# Byte code execution (2/3)

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

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

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

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

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

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

# Byte code execution (3/3)

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

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

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

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

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 they describe properties, e.g.,

- the set $\{1, 3, 5, \dots\}$ describes the property $odd$
- the set $\{2, 4, 6, \dots\}$ 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 $[4; 10]$
- $\dots$

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 forms 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

# Collecting semantics, revisited (3/3)

A post-fixed point $\Sigma'$ of $T(\Sigma) = I \cup \{s' \mid \exists s \in \Sigma : s \to s'\}$ satisfies:

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

Thus $\Sigma'$ is an *invariant*.

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

Note: any post-fixed point of $T$ 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:

$$
\begin{array}{cl}
\wp(S^*) & \text{Partial traces} \\
\alpha^* \big\downarrow \big\uparrow \gamma^* & \\
\wp(S \times S) & \text{Reflexive, transitive closure} \\
\alpha^\bullet \big\downarrow \big\uparrow \gamma^\bullet & \\
\wp(S) & \text{Reachable states}
\end{array}
$$

# Stronger properties, stronger collecting semantics

There is a hierarchy of increasingly powerful collecting semantics:

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

Each can be expressed as a least fixed point

# Example: collecting semantics for the CE

Reachable states: $T(\Sigma) = I \cup \{s' \mid \exists s \in \Sigma : s \to s'\}$

Recall transitions:

$$\langle \texttt{(}\mathtt{t}_0\,\mathtt{t}_1\,\texttt{c)}, r\rangle \longrightarrow \langle \texttt{e}, r'[\texttt{x} \mapsto w][\texttt{k} \mapsto w_{\texttt{c}}]\rangle$$

if $[\texttt{(}\lambda\ \texttt{(x k) e)}, r'] = \mu_t(\mathtt{t}_0, r)$

$w = \mu_t(\mathtt{t}_1, r)$

$w_{\texttt{c}} = \mu_c(\texttt{c}, r)$

$$\langle \texttt{(c t)}, r\rangle \longrightarrow \langle \texttt{e}, r'[\texttt{v} \mapsto w]\rangle$$

if $[\texttt{(}\lambda\ \texttt{(v) e)}, r'] = \mu_c(\texttt{c}, r)$

$w = \mu_t(\texttt{t}, r)$

# Example: collecting semantics for the CE

Reachable states: $T(\Sigma) = I \cup \{s' \mid \exists s \in \Sigma : s \to s'\}$

Recall transitions:

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

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

Direct definition (for a given program $\texttt{(}\lambda\ \texttt{(k) e)}$):

$$\begin{aligned}
T(S) = I_{\texttt{(}\lambda\ \texttt{(k) e)}} \\
&\cup \{\langle \mathtt{e}', r'[\mathtt{x} \mapsto w][\mathtt{k}' \mapsto w_{\mathtt{c}}]\rangle \mid \exists \langle \texttt{(}\mathtt{t}_0\,\mathtt{t}_1\,\mathtt{c}\texttt{)}, r\rangle \in S : \\
&\qquad [\texttt{(}\lambda\ \texttt{(x k}'\texttt{) e}'\texttt{)}, r'] \in \mu_t^{\wp}(\mathtt{t}_0, \{r\}) \\
&\qquad \wedge\ w \in \mu_t^{\wp}(\mathtt{t}_1, \{r\}) \\
&\qquad \wedge\ w_{\mathtt{c}} \in \mu_c^{\wp}(\mathtt{c}, \{r\})\} \\
&\cup \{\langle \mathtt{e}', r'[\mathtt{v} \mapsto w]\rangle \mid \exists \langle \texttt{(c t)}, r\rangle \in S : \\
&\qquad [\texttt{(}\lambda\ \texttt{(v) e}'\texttt{)}, r'] \in \mu_c^{\wp}(\mathtt{c}, \{r\}) \\
&\qquad \wedge\ w \in \mu_t^{\wp}(\mathtt{t}, \{r\})\}
\end{aligned}$$

# Example cont.: properties of helper functions

Recall helper functions:

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

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

Collecting helper functions:

$$\mu_t^{\wp} : \mathit{TExp} \times \wp(\mathit{Env}) \to \wp(\mathit{Val})$$
$$\mu_t^{\wp}(\mathtt{x}, E) = \{r(\mathtt{x}) \mid r \in E\}$$
$$\mu_t^{\wp}(\mathtt{v}, E) = \{r(\mathtt{v}) \mid r \in E\}$$
$$\mu_t^{\wp}(\mathtt{(}\lambda\ \mathtt{(x\ k)\ e)}, E) = \{[\mathtt{(}\lambda\ \mathtt{(x\ k)\ e)}, r] \mid r \in E\}$$

$$\mu_c^{\wp} : \mathit{CExp} \times \wp(\mathit{Env}) \to \wp(\mathit{Val})$$
$$\mu_c^{\wp}(\mathtt{k}, E) = \{r(\mathtt{k}) \mid r \in E\}$$
$$\mu_c^{\wp}(\mathtt{(}\lambda\ \mathtt{(v)\ e)}, E) = \{[\mathtt{(}\lambda\ \mathtt{(v)\ e)}, r] \mid r \in E\}$$

# 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.