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:

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

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?

IMP semantics

IMP programs

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

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

$$
\begin{array}{lll}
\mathit{test} \ni \mathit{BExp} ::= e \ \mathit{comp} \ e & \qquad \text{where} \quad \mathit{comp} \in \{=, <>, <, \ldots\} \\
\quad \mid \mathit{test} \texttt{ and } \mathit{test} & \\
\quad \mid \mathit{test} \texttt{ or } \mathit{test} &
\end{array}
$$

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

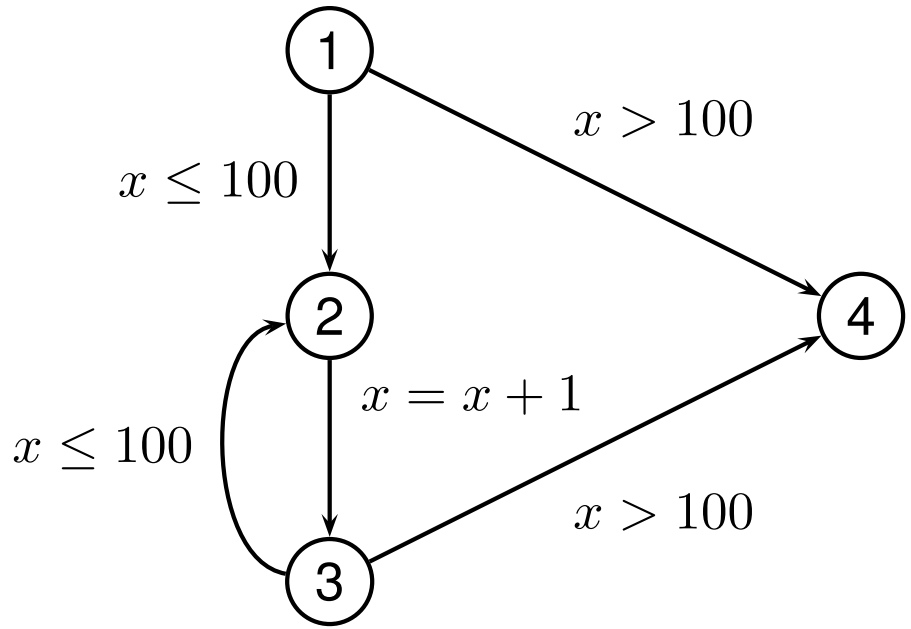
Imperative programs as flow graphs

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

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

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

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

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

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

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

Semantics of arithmetic expressions and tests

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

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

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

$$\mathcal{B} : BExp \to Store \to \wp(\mathbb{B}) \quad \text{where} \quad \mathbb{B} = \{true, false\}$$
$$\mathcal{B}\,[\![e\ comp\ e']\!]\,\rho = \{true \mid n \in \mathcal{A}\,[\![e]\!]\,\rho \ \wedge\ n' \in \mathcal{A}\,[\![e]\!]\,\rho \ \wedge\ n\ comp\ n'\} \bigcup \{false \mid n \in \mathcal{A}\,[\![e]\!]\,\rho \ \wedge\ n' \in \mathcal{A}\,[\![e]\!]\,\rho \ \wedge\ \neg(n\ comp\ n')\}$$
$$\mathcal{B}\,[\![test\ \mathtt{and}\ test']\!]\,\rho = \{b \ \wedge\ b' \mid b \in \mathcal{B}\,[\![test]\!]\,\rho \ \wedge\ b' \in \mathcal{B}\,[\![test']\!]\,\rho\}$$
$$\mathcal{B}\,[\![test\ \mathtt{or}\ test']\!]\,\rho = \{b \ \vee\ b' \mid b \in \mathcal{B}\,[\![test]\!]\,\rho \ \wedge\ b' \in \mathcal{B}\,[\![test']\!]\,\rho\}$$

IMP program execution as a transition system

States are pairs:

$$State = V \times Store$$

There is one case per instruction:

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

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

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:

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

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

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

Bytecode semantics

Scope, mutation, and semantics

The CPS semantics 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

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

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.

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

Virtual machine domains

$$loc \ni Locations \qquad \text{(some countable number of locations)}$$

$$v \ni Value = n \mid loc \mid \texttt{null}$$

$$s \ni OperandStack = Value^*$$

$$l \ni LocalVar = [Value_\perp]$$

$$Frame = Method \times Address \times LocalVar \times OperandStack$$

$$sf \ni CallStack = Frame^*$$

$$o \ni Object = Class \times (FieldName \rightharpoonup Value)$$

$$h \ni Heap = Locations \to Object_\perp$$

$$State = Heap \times CallStack$$

We now define a number of shorthands and helper functions:

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

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

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

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

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)

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

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

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

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

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

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

Byte code execution (3/3)

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

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

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

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

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 $\{1, 3, 5, \ldots\}$ 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 $[4; 10]$
- ...

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

Collecting semantics, revisited (3/3)

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

- $I \subseteq \Sigma' \sim$ "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 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:

$$\wp(S^*) \qquad \lambda X.\, \{s \mid s \in S\} \cup \{\sigma s s' \mid \sigma s \in X \ \wedge \ s \to s'\}$$

$$\alpha^* \downarrow \quad \uparrow \gamma^*$$

$$\wp(S \times S) \qquad \lambda Y.\, \{\langle s,\, s\rangle \mid s \in S\} \cup \{\langle s,\, s''\rangle \mid \exists s' : \langle s,\, s'\rangle \in Y \ \wedge \ s' \to s''\}$$

$$\alpha^\bullet \downarrow \quad \uparrow \gamma^\bullet$$

$$\wp(S) \qquad \lambda Z.\, I \cup \{s' \mid \exists s \in Z : s \to s'\}$$

Each can be expressed as a least fixed point