Looking back, looking forward

This is the last lecture using IMP (hooray!). Done:
- Abstract syntax
- Operational semantics (large-step and small-step)
- Semantic properties of (sets of) programs
- “Pseudo-denotational” semantics

Now:
- Packet-filter languages and other examples
- Equivalence of programs in a semantics
- Equivalence of different semantics

Next lecture: Local variables, lambda-calculus

Packet Filters

A very simple view of packet filters:
- Some bits come in off the wire
- Some application(s) want the “packet” and some do not (e.g., port number)
- For safety, only the O/S can access the wire
- For extensibility, the applications accept/reject packets

Conventional solution goes to user-space for every packet and app that wants (any) packets

Faster solution: Run app-written filters in kernel-space

Language-based approaches

1. Interpret a language
   + clean operational semantics, + portable, - may be slow (+ filter-specific optimizations), - unusual interface

2. Translate a language into C/assembly
   + clean denotational semantics, + employ existing optimizers, - upfront cost, - unusual interface

3. Require a conservative subset of C/assembly
   + normal interface, - too conservative w/o help

IMP has taught us about (1) and (2) — we’ll get to (3)
A General Pattern

Packet filters move the code to the data rather than data to the code

General reasons: performance, security, other?

Other examples:
- Query languages
- Active networks
- Client-side web scripts (Javascript)

Equivalence motivation

- Program equivalence (we change the program):
  - code optimizer
  - code maintainer

- Semantics equivalence (we change the language):
  - interpreter optimizer
  - language designer
    - (prove properties for equivalent semantics with easier proof)

Note: Proofs may seem easy with the right semantics and lemmas
- (almost never start off with right semantics and lemmas)

Note: Small-step operational semantics often has harder proofs, but models more interesting things

What is equivalence?

Equivalence depends on what is observable!
- Partial I/O equivalence (if terminates, same ans)
  - while 1 skip equivalent to everything
  - not transitive
- Total I/O equivalence (same termination behavior, same ans)
- Total heap equivalence (same termination behavior, same heaps)
  - All (almost all?) variables have the same value
- Equivalence plus complexity bounds
  - Is \( O(2^n) \) really equivalent to \( O(n) \)?
  - Is “runs within 10ms of each other” important?
- Syntactic equivalence (perhaps with renaming)
  - Too strict to be interesting?

In PL, equivalence most often means total I/O equivalence

Example

Are the following programs equivalent?

\[
\begin{align*}
\text{C: } & \quad x:=5; \\
& \quad \text{popheap x; } \\
& \quad y:=4; \\
& \quad \text{pushheap; } \\
& \quad x:=3; \\
& \quad \text{popheap y; } \\
& \quad \text{ans := x}
\end{align*}
\]

\[
\begin{align*}
\text{H: } & \quad \text{x := 5; } \\
& \quad \text{popheap x; } \\
& \quad \text{y := 4; } \\
& \quad \text{pushheap; } \\
& \quad \text{x := 3; } \\
& \quad \text{popheap y; } \\
& \quad \text{ans := x}
\end{align*}
\]

What about these two programs?

\[
\begin{align*}
\text{C: } & \quad x:=5; \\
& \quad \text{popheap x;} \\
& \quad y:=4; \\
& \quad \text{pushheap; } \\
& \quad x:=3; \\
& \quad \text{popheap x; } \\
& \quad \text{ans := x}
\end{align*}
\]

\[
\begin{align*}
\text{H: } & \quad \text{x := 5; } \\
& \quad \text{popheap x;} \\
& \quad \text{y := 4; } \\
& \quad \text{pushheap; } \\
& \quad \text{x := 3; } \\
& \quad \text{popheap x; } \\
& \quad \text{ans := x}
\end{align*}
\]

Program Example: Strength Reduction

Motivation: Strength reduction
- A common compiler optimization due to architecture issues

Theorem: \( H ; e \downarrow c \) if and only if \( H ; e + e \downarrow c \)

Proof sketch:
- Prove separately for each direction
- Invert the assumed derivation, use hypotheses plus a little math to derive what we need
- Hmm, doesn’t use induction. That’s because this theorem isn’t very useful...

Program Example: Nested Strength Reduction

Theorem: If \( e' \) has a subexpression of the form \( e * 2 \), then \( H ; e' \downarrow c' \) if and only if \( H ; e'' \downarrow c' \)
where \( e'' \) is \( e' \) with \( e * 2 \) replaced with \( e + e \)

First some useful metanotation:

\[
C ::= [ ] | C + e | e + C | C * e | e * C
\]

\( C[e] \) is “\( C \) with \( e \) in the hole” (inductive definition of “stapling”)

Crisper statement of theorem:

\[
H ; C[e * 2] \downarrow c' \text{ if and only if } H ; C[e + e] \downarrow c'
\]

Proof sketch: By induction on structure (“syntax height”) of \( C \)
- The base case (\( C = [ ] \)) follows from our previous proof
- The rest is a long, tedious, (and instructive!) induction
Proof reuse

As we cannot emphasize enough, proving is just like programming.

The proof of nested strength reduction had nothing to do with e * 2 and e + e except in the base case where we used our previous theorem.

A much more useful theorem would parameterize over the base case so that we could get the “nested X” theorem for any appropriate X:

\[(H; e_1 \downarrow c) \text{ if and only if } H; e_2 \downarrow c,\]
then \([H; C[e_1] \downarrow c'] \text{ if and only if } H; C[e_2] \downarrow c']\)

The proof is identical except the base case is “by assumption”.

Small-step program equivalence

These sort of proofs also work with small-step semantics (e.g., our IMP statements), but tend to be more cumbersome, even to state.

Example: The statement-sequence operator is associative. That is,

(a) For all n, if \(H; s_1; s_2; s_3 \rightarrow^n H' ; \text{skip}\) then there exist \(H'' \text{ and } n'\) such that \(H; (s_1; s_2); s_3 \rightarrow^{n'} H'' ; \text{skip}\) and \(H''(\text{ans}) = H'(\text{ans})\).

(b) If for all n there exist \(H'\) and \(s'\) such that \(H; s_1; s_2; s_3 \rightarrow^n H' ; s'\), then for all n there exist \(H'' \text{ and } s''\) such that \(H; (s_1; s_2); s_3 \rightarrow^n H'' ; s''\).

(Proof needs a much stronger induction hypothesis.)

One way to avoid it: Prove large-step and small-step semantics equivalent, then prove program equivalences in whichever is easier.

Language Equivalence Example

**IMP w/o multiply large-step:**

\[
\begin{array}{c|c|c}
\text{CONST} & \text{VAR} & \text{ADD} \\
\hline
H; c \downarrow c & H; x \downarrow H(x) & H; e_1 \downarrow c_1, H; e_2 \downarrow c_2 \\
\end{array}
\]

**IMP w/o multiply small-step:**

\[
\begin{array}{c|c|c}
\text{SVAR} & \text{SADD} & \text{SLEFT} & \text{SRIGHT} \\
\hline
H; x \rightarrow H(x) & H; c_1 + c_2 \rightarrow c_1 + c_2 \\
\end{array}
\]

Theorem: Semantics are equivalent: \(H; e \downarrow c\) if and only if \(H; e \rightarrow^* c\).

Proof: We prove the two directions separately...

Proof, part 1

First assume \(H; e \downarrow c\) and show \(\exists n. H; e \rightarrow^n c\).

Lemma (prove it!): If \(H; e \rightarrow^n e'\), then \(H; e_1 + e \rightarrow^n e_1 + e'\) and \(H; e_2 \rightarrow^n e' + e_2\).

Ex. Proof by induction on n:

- PROOF: Inductive case uses SLEFT and SRIGHT.

Given the lemma, prove by induction on derivation of \(H; e \downarrow c\):

- PROOF: Add: Derivation with ADD implies e = e_1 + e_2, c = c_1 + c_2.

Proof, part 2

Now assume \(\exists n. H; e \rightarrow^n c\) and show \(H; e \downarrow c\).

Proof by induction on n:

- PROOF: n = 0: e is c and CONST lets us derive \(H; c \downarrow c\).

- n > 0: (Clever: break into first step and remaining ones) \(\exists e', H; e \rightarrow e'\) and \(H; e' \rightarrow^{n-1} c\).

So by our lemma \(H; e \rightarrow e'\).

So this lemma suffices: If \(H; e \rightarrow e'\) and \(H; e' \downarrow c\), then \(H; e \downarrow c\).

Proof the lemma by induction on derivation of \(H; e \rightarrow e'\):

- PROOF: SVAR: ...
- PROOF: SADD: ...
- PROOF: SLEFT: ...
- PROOF: SRIGHT: ...
Part 2, key lemma

Lemma: If $H; e \rightarrow e'$ and $H; e' \Downarrow c$, then $H; e \Downarrow c$.

Prove the lemma by induction on derivation of $H; e \rightarrow e'$:

- **SVAR**: Derivation with **SVAR** implies $e$ is some $x$ and $e' = H(x) = c$, so derive, by **VAR**, $H; x \Downarrow H(x)$.
- **SADD**: Derivation with **SADD** implies $e$ is some $c_1 + c_2$ and $e' = c_1 + c_2 = c$, so derive, by **ADD** and two **CONST**, $H; c_1 + c_2 \Downarrow c_1 + c_2$.
- **SLEFT**: Derivation with **SLEFT** implies $e = e_1 + e_2$ and $e' = e_1' + e_2$ and $H; e_1 \rightarrow e_1'$ for some $e_1, e_2, e_1'$. Since $e' = e_1' + e_2$ inverting assumption $H; e' \Downarrow c$ gives $H; e_1' \Downarrow c_1$, $H; e_2 \Downarrow c_2$ and $c = c_1 + c_2$. Applying the induction hypothesis to $H; e_1 \rightarrow e_1'$ and $H; e_1' \Downarrow c_1$ gives $H; e_1 \Downarrow c_1$. So use **ADD**, $H; e_1 \Downarrow c_1$, and $H; e_2 \Downarrow c_2$ to derive $H; e_1 + e_2 \Downarrow c_1 + c_2$.
- **SRIGHT**: Analogous to **SLEFT**.

The cool part, redux

Step through the **SLEFT** case more visually:

By assumption, we must have derivations that look like this:

$$H; e_1 \rightarrow e_1' \quad H; e_1' \Downarrow c_1 \quad H; e_2 \Downarrow c_2$$

Now go grab the one hypothesis we haven’t used yet and combine it with our inductive result to derive our answer:

$$H; e_1 \Downarrow c_1 \quad H; e_2 \Downarrow c_2$$

Conclusions

- Equivalence is a subtle concept
- Proofs “seem obvious” only when the definitions are right
- Some other language-equivalence claims:

Replace **WHILE** rule with

$$H; e \Downarrow c \quad c \leq 0 \quad H; e \Downarrow c \quad c > 0$$

Equivalent to our original language

Change syntax of heap and replace **ASSIGN** and **VAR** rules with

$$H; x := e \rightarrow H, x \Rightarrow e \Downarrow \text{skip} \quad H; H(x) \Downarrow c \quad H; x \Downarrow c$$

NOT equivalent to our original language