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

What we need

Now the O/S writer is defining the packet-filter language!

Properties we wish of (untrusted) filters:

1. Do not corrupt kernel data structures
2. Terminate (within a time bound)
3. Run fast (the whole point)

Should we download some C/assembly code? (Get 1 of 3)

Should we make up a language and “hope” it has these properties?

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

Program Example: Strength Reduction

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

Theorem: \(H ; e \ast 2 \Downarrow c\) 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 \(e \ast 2\),
then \(H ; e \Downarrow c\) if and only if \(H ; e'' \Downarrow c'\)
where \(e''\) is \(e\) with \(e \ast 2\) replaced with \(e + e\)

First some useful metanotation:

\[C ::= [\cdot] \mid C + e \mid e + C \mid C \ast e \mid e \ast C\]

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

Crisper statement of theorem:

\(H ; C[e \ast 2] \Downarrow c\) if and only if \(H ; C[e + e] \Downarrow c\)

Proof sketch: By induction on structure (“syntax height”) of \(C\)
- The base case \((C = [\cdot])\) 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 \(e \ast 2\) 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 \(X\):

If \((H ; e_1 \Downarrow c\) if and only if \(H ; e_2 \Downarrow c\)),
then \((H ; C[e_1] \Downarrow c\) 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 \(H ; s_1; s_2; s_3 \rightarrow^n H'\) and \(s'\) such that \(H'' \rightarrow^n H'' ; s\) and \(H''(ans) = H'(ans)\).

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

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

(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:

<table>
<thead>
<tr>
<th></th>
<th>VAR</th>
<th>SADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>H; e ⇓ c</td>
<td></td>
<td>H; e1 + e2 ⇓ c1 + c2</td>
</tr>
</tbody>
</table>

IMP w/o multiply small-step:

<table>
<thead>
<tr>
<th></th>
<th>SVAR</th>
<th>SLEFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>H; s → H(x)</td>
<td></td>
<td>H; e1 + e2 → e1′ + e2</td>
</tr>
</tbody>
</table>

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

Proof: We prove the two directions separately...

Part 1, continued

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 \( H; e_1 + e \rightarrow^n e_1 + e' \)

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

- ADD: Derivation with ADD implies \( e = e_1 + e_2 \) and \( e' = e_1' + e_2' \)

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 \( H; e_1 + e \rightarrow^n e_1 + e' \)

- Proof by induction on \( n \)
  - Inductive case uses SLEFT and SRIGHT

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

- CONST: Derivation with CONST implies \( e = c \), and we can derive \( H; e \rightarrow^0 c \)
  - VAR: Derivation with VAR implies \( e = x \) for some \( x \) where \( H(x) = c \), so derive \( H; e \rightarrow^1 c \) with SVAR
  - ADD: ...

Proof, part 2

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

Proof by induction on \( n \):

- \( n = 0 \): \( e \) is \( c \) and CONST lets us derive \( H; c \downarrow c \)
- \( n > 0 \): (Clever: break into first step and remaining ones) \( H; e \rightarrow e' \) and \( H; e' \rightarrow^{n-1} c \).

- Prove the lemma by induction on derivation of \( H; e \rightarrow e' \):
  - SVAR: ...
  - SADD: ...
  - SLEFT: ...
  - 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 \( e = e_1 + e_2 \) for some \( e \)
  - SADD: Derivation with SADD implies \( e = c_1 + c_2 \) and \( e' = c_1' + c_2' \), and \( e' = e_1' + e_2' \)

The cool part, redux

Step through the SLEFT case more visually:

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

\[
\begin{align*}
H; e_1 & \rightarrow e_1' \\
H; e_1 + e_2 & \rightarrow e_1' + e_2 \\
H; e_1' + e_2 & \downarrow c_1 + c_2
\end{align*}
\]

Grab the hypothesis from the left and the left hypothesis from the right and use induction to get \( H; e_1 \downarrow c_1 \)

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

\[
\begin{align*}
H; e_1 & \downarrow c_1 \\
H; e_1 + e_2 & \downarrow c_1 + c_2
\end{align*}
\]
A nice payoff

Theorem: The small-step semantics is deterministic:

if $H; e \rightarrow^* c_1$ and $H; e \rightarrow^* c_2$, then $c_1 = c_2$

Not obvious (see sleft and sright), nor do I know a direct proof

Given $((1 + 2) + (3 + 4)) + (5 + 6)) + (7 + 8)$ there are many execution sequences, which all produce 36 but with different intermediate expressions

Proof:

- Large-step evaluation is deterministic (easy induction proof)
- Small-step and and large-step are equivalent (just proved that)
- So small-step is deterministic
- Convince yourself a deterministic and a nondeterministic semantics cannot be equivalent

Conclusions

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

Replace WHILE rule with

$H; e \downarrow c \quad c \leq 0$
$H; \text{while } e \text{ } s \text{ } \rightarrow \text{ } H; \text{skip}$
$H; e \downarrow c \quad c > 0$
$H; \text{while } e \text{ } s \text{ } \rightarrow \text{ } H; \text{ } s; \text{while } e \text{ } s$

Equivalent to our original language

Change syntax of heap and replace ASSIGN and VAR rules with

$H; x := e \rightarrow H, x \mapsto e; \text{skip}$
$H; H(x) \downarrow c$
$H; x \downarrow c$

NOT equivalent to our original language