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?
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)
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 $X$:

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 $H; s_1; (s_2; s_3) \rightarrow^n H'; skip$ then there exist $H''$ and $n'$ such that $H; (s_1; s_2); s_3 \rightarrow^{n'} H''; skip$ and $H''(ans) = H'(ans)$.

(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 $H''$ 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:

\[
\begin{array}{l}
\text{CONST} & \quad \text{VAR} & \quad \text{ADD} \\
H; c \Downarrow c & \quad H; x \Downarrow H(x) & \quad H; e_1 \Downarrow e_1, e_2 \Downarrow c_2 \\
\end{array}
\]

IMP w/o multiply small-step:

\[
\begin{array}{l}
\text{SVAR} & \quad \text{SADD} \\
H; x \rightarrow H(x) & \quad H; c_1 + c_2 \rightarrow c_1 + c_2 \\
\end{array}
\]

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 \( H; e + e_2 \rightarrow^n e' + e_2 \).

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

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

- \( \text{CONST} \): Derivation with \( \text{CONST} \) implies \( e = c \), and we can derive \( H; c \rightarrow^0 c \)
- \( \text{VAR} \): Derivation with \( \text{VAR} \) implies \( e = x \) for some \( x \) where \( H(x) = c \), so derive \( H; e \rightarrow^1 c \) with SVAR
- \( \text{ADD} \): ...
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
    \[
    \begin{align*}
    H; e \downarrow c & \quad c \leq 0 \\
    H; \text{while } e s & \rightarrow H; \text{skip} \\
    H; e \downarrow c & \quad c > 0 \\
    H; \text{while } e s & \rightarrow H; s; \text{while } e s
    \end{align*}
    \]
  - Equivalent to our original language
  - Change syntax of heap and replace ASSIGN and VAR rules with
    \[
    \begin{align*}
    H; x := e & \rightarrow H, x \mapsto e; \text{skip} \\
    H; H(x) \downarrow c & \quad H; x \downarrow c
    \end{align*}
    \]
  - NOT equivalent to our original language