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)

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
What we need
Now the O/S writer is defining the packet-filter language!
Properties we wish of (untrusted) filters:
1. Don’t 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?

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 (Javascrip)
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: $H; e * 2 \downarrow c$ 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 $e * 2$,
then $H; e' \downarrow c'$ if and only if $H; e'' \downarrow c'$
where $e''$ is $e'$ with $e * 2$ replaced by $e + e$

First some useful metanotation:

$$C ::= [\cdot] \mid C + e \mid e + C \mid C * e \mid e * C$$

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

Crisper statement of theorem:

$$H; C[e * 2] \downarrow c'$$

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 $e * 2$ 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 $H; s_1; (s_2; s_3) \rightarrow^n H'; \text{skip}$
then there exist $H''$ 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 $H''$ 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:

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

IMP w/o multiply small-step:

\[
\begin{array}{ccc}
\text{SVAR} & \text{SADD} & \text{SLEFT} \\
H ; x \rightarrow H(x) & H ; e_1 + e_2 \rightarrow e_1 + e_2 & H ; e_1 \rightarrow e_1, H ; e_2 \rightarrow e_2 \\
\end{array}
\]

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

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

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

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

**Proof, part 2**

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

Proof by induction on \( n \):

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

**The cool part, redux**

Step through the \( \text{SLEFT} \) case more visually:

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

\[
\begin{array}{ccc}
H ; e_1 \rightarrow e_1' & H ; e_1 \downarrow e_1, H ; e_2 \downarrow e_2, H ; e_1 + e_2 \downarrow c_1 + c_2 \\
H ; e_1 + e_2 \rightarrow e_1' + e_2 & H ; e_1 + e_2 \downarrow c_1 + c_2 \\
\end{array}
\]

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:

\[
\begin{array}{ccc}
H ; e_1 \downarrow c_1, H ; e_2 \downarrow c_2 & H ; e_1 + e_2 \downarrow c_1 + c_2 \\
\end{array}
\]
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 \texttt{sleft} and \texttt{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 large-step are equivalent (just proved that)
- So small-step is deterministic
- Convince yourself a deterministic and a nondeterministic semantics can’t be equivalent

Conclusions

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

Replace \texttt{WHILE} rule with
\[
\begin{align*}
H; e \downarrow c & \quad c \leq 0 \quad \\
H; \text{while } e \textsf{ s} & \rightarrow H; \text{ skip} \quad \\
H; e s & \rightarrow H; s; \text{ while } e s
\end{align*}
\]
Equivalent to our original language

Change syntax of heap and replace \texttt{ASSIGN} and \texttt{VAR} rules with
\[
\begin{align*}
H; x := e & \rightarrow H, x \mapsto e; \text{skip} \\
H; H(x) \downarrow c & \rightarrow H; x \downarrow c
\end{align*}
\]
\texttt{NOT} equivalent to our original language