CSE 505: Concepts of Programming Languages

Dan Grossman Fall 2005 Lecture 5— Little Trusted-Languages; Equivalence

Where are we

Today is IMP's last day (hooray!). Done:

- Abstract Syntax
- Operational Semantics (large-step and small-step)
- "Denotational" Semantics
- Semantic properties of (sets of) programs

Today:

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

Next time: Local variables, lambda-calculus

Packet Filters

Almost everything I know about packet filters:

- Some bits come in off the wire
- Some application(s) want the "packet" and some do not (e.g., port number)
- $\bullet\,$ For safety, only the O/S can access the wire.
- For extensibility, only an application can accept/reject a packet.

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. 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?

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)

<u>A General Pattern</u>

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

Equivalence motivation

- Program equivalence (change program): code optimizer, code maintainer
- Semantics equivalence (change language): interpreter optimizer, language designer (prove properties for equivalent semantics with easier proof)
- Both: Great practice for strengthening inductive hypothesis (you will do this again in grad school)

Warning: Proofs are easy with the right semantics and lemmas Note: Small-step 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 (same termination behavior, same ans)
- Total heap equivalence (at termination, all (almost all) variables have the same value)
- Equivalence plus complexity bounds
 - Is $O(2^{n^n})$ really equivalent to O(n)?
- Syntactic equivalence (perhaps with renaming)
 - too strict to be interesting

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: Just need "inversion of derivation" and math (hmm, no induction).

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 * 2replaced with 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".

So: If $(e_1 = C[e * 2]$ and $e_2 = C[e + e])$, then $(H; e_1 \Downarrow c'$ if and only if $H; e_2 \Downarrow c')$.

Proof sketch: By induction on structure ("syntax height") of C.

Small-step program equivalence

Theorem and proof significantly simplified by:

- Determinism
- Termination
- Large-step semantics

IMP statements have only determinism.

Theorem: 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 $H ; (s_1; s_2); s_3 \rightarrow^n H''; s''$.

<u>continued</u>

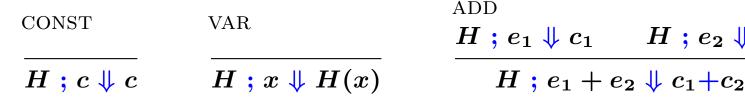
Lemma: For all *n*, if H; s_1 ; $(s_2; s_3) \rightarrow^n H'$; s', then either (1) s' has the form s'_1 ; $(s_2; s_3)$ and H; $(s_1; s_2); s_3 \rightarrow^n H'$; $(s'_1; s_2); s_3$ or (2) H; $(s_1; s_2); s_3 \rightarrow^n H'; s'$.

Lemma implies theorem: It's stronger because if s' is **skip**, then only (2) applies and we have H'' = H' and n' = n.

Proof of lemma: Tedious (will post for the curious).

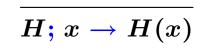
Language Equivalence Example

IMP w/o multiply:



IMP w/o multiply small-step:

SVAR



SLEFT

$$H; e_1 \to e'_1$$
$$H; e_1 + e_2 \to e'_1 + e_2$$

ADD

SADD

$$H; c_1 + c_2 \rightarrow c_1 + c_2$$

SRIGHT $H; e_2 \rightarrow e'_2$ $\overline{H; e_1 + e_2 \rightarrow e_1 + e_2'}$

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

Theorem: Semantics are equivalent, i.e., $H ; e \Downarrow c$ if and only if $H; e \rightarrow^* c$.

Proof: We prove the two directions separately.

Proof, part 1:

First assume $H ; e \Downarrow c$; 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 uses SLEFT and SRIGHT.) Given the lemma, prove by induction on height h of derivation of $H ; e \Downarrow c$:

- h = 1: Derivation is via CONST (so $H; e \rightarrow^{0} c$) or VAR (so $H; e \rightarrow^{1} c$).
- h > 1: Derivation ends with ADD, so e has the form $e_1 + e_2$, $H ; e_1 \Downarrow c_1, H ; e_2 \Downarrow c_2$, and c is $c_1 + c_2$. By induction $\exists n_1, n_2$. $H; e_1 \rightarrow^{n_1} c_1$ and $H; e_2 \rightarrow^{n_2} c_2$. So by our lemma $H; e_1 + e_2 \rightarrow^{n_1} c_1 + e_2$ and $H; c_1 + e_2 \rightarrow^{n_2} c_1 + c_2$. So SADD lets us derive $H; e_1 + e_2 \rightarrow^{n_1+n_2+1} c$.

Proof, part 2:

Now assume $\exists n. H; e \rightarrow^n c$; show $H; e \Downarrow c$. By induction on n:

- n = 0: e is c and CONST lets us derive H; $c \Downarrow c$.
- n > 0: ∃e'. H; e → e' and H; e' →ⁿ⁻¹ c. By induction H ; e' ↓ c. So this lemma suffices: If H; e → e' and H ; e' ↓ c, then H ; e ↓ c.
 Prove the lemma by induction on height h of derivation of H; e → e':
 - -h = 1: Derivation ends with SVAR (so e' = c = H(x) and VAR gives $H ; x \Downarrow H(x)$) or with SADD (so e is some $c_1 + c_2$ and $e' = c = c_1 + c_2$ and ADD gives $H ; c_1 + c_2 \Downarrow c_1 + c_2$).

-h > 1: Derivation ends with SLEFT or SRIGHT ...

Proof, part 2 continued:

If e has the form $e_1 + e_2$ and e' has the form $e'_1 + e_2$, then the assumed derivations end like this:

$$\frac{H; e_1 \rightarrow e_1'}{H; e_1 + e_2 \rightarrow e_1' + e_2} \qquad \qquad \frac{H; e_1' \Downarrow c_1 \qquad H; e_2 \Downarrow c_2}{H; e_1' + e_2 \Downarrow c_1 + c_2}$$

Using $H; e_1 \rightarrow e'_1, H; e'_1 \Downarrow c_1$, and the induction hypothesis, $H; e_1 \Downarrow c_1$. Using this fact, $H; e_2 \Downarrow c_2$, and ADD, we can derive $H; e_1 + e_2 \Downarrow c_1 + c_2$.

(If e has the form $e_1 + e_2$ and e' has the form $e_1 + e'_2$, the argument is analogous to the previous case (prove it!).)

Conclusions

- Equivalence is a subtle concept.
- Proofs "seem obvious" only when the definitions are right.
- Some other language-equivalence claims: Replace WHILE rule with

 $\frac{H \; ; e \Downarrow c \quad c \leq 0}{H \; ; \text{ while } e \; s \to H \; ; \, \text{skip}} \qquad \frac{H \; ; e \Downarrow c \quad c > 0}{H \; ; \, \text{ while } e \; s \to H \; ; \; s ; \, \text{while } e \; s}$

Theorem: Languages are equivalent. (True) Change syntax of heap and replace ${\rm ASSIGN}$ and ${\rm VAR}$ rules with

 $\frac{H \ ; H(x) \Downarrow c}{}$

 $H \ ; x := e \rightarrow H, x \mapsto e \ ; ext{skip} \qquad H \ ; x \Downarrow c$

Theorem: Languages are equivalent. (False)