CS 411 Lecture 6 Command equivalences and program transformations September 10, 2004
Lecturer: Radu Rugina

We can use our semantic models to precisely characterize equivalences of programs. For instance, using the large-step evaluation relation, two commands c, c' are equivalent if, starting with the same initial state, they yield the same final state:

$$\forall s, s' : \langle c, s\rangle \Downarrow s' \iff \langle c, s\rangle \Downarrow s'$$

Such equivalences allow us to show that program transformations and optimizations preserve the semantics of the original program. In other words, we can use our formal machinery to prove the correctness of program transformations.

Let's consider a couple of standard compiler optimizations. Copy propagation is an optimization that identifies pairs variables that hold the same value as the result of a copy assignment; the optimization then replaces occurrences of one variable with the other. Such a transformation is correct if the resulting program is equivalent to the original one:

$$x := t; c \sim x := t; c[t/x]$$

where $c[t/x]$ is the substitution of variable t for x in command c (i.e., each occurrence of x gets replaced with t). But this transformation holds only if x or t do not get assigned new values in c; otherwise, the invariant $x = t$ is not guaranteed to hold and the transformation may be unsafe. Hence, we need to impose the following safety condition:

$$x, t \notin Wr(c)$$

where $Wr(c)$ is the set of variables being assigned in c. Note that this condition is safe, but conservative (why don't we use a more accurate condition expressed using program states?).

Similarly, constant propagation identifies variables that hold constant values and replaces occurrences of those variables with their values:

$$x := n; c \sim x := n; c[n/x] \qquad \text{if } x \notin Wr(c)$$

Such transformations give opportunities to the compiler to perform dead code elimination – removing assignments whose effects are not being used in the rest of the program:

$$x := e; c \sim_x c \qquad \text{if } x \notin Rd(c)$$

where $Rd(c)$ is the set of variables being read by command c. However, we must rephrase our notion of equivalence here: the transformed program and the original one yield the same final states except for the value of variable x. We write this equivalence as $\sim_x$.

A standard loop optimization is loop invariant code motion, which hoists assignments out of the loop body to the point right before the loop:

$$\textbf{while } b \textbf{ do } c1; x := e; c2 \sim_x x := e; \textbf{while } b \textbf{ do } c1; c2$$

What would be an appropriate condition that would ensure the safety of this transformation? Why do we use the $\sim_x$ equivalence?

We would like to have a clear definition for what substitution or read/write sets mean. An easy way to define these is by induction on the structure of commands and expressions. For instance, substitution can be inductively defined as follows:

c	$c[e/x]$
skip	**skip**
$y := e'$	$y := e'[e/x]$
$c1; c2$	$c1[e/x]; c2[e/x]$
if b **then** $c1$ **else** $c2$	**if** $b[e/x]$ **then** $c1[e/x]$ **else** $c2[e/x]$
while b **do** c	**while** $b[e/x]$ **do** $c[e/x]$

where the substitution for boolean and arithmetic expressions is:

e'	$e'[e/x]$
n	n
x	e
y	$y (if y \neq x)$
$e1 ope 2$	$e1[e/x] ope 2[e/x]$

The sets of variables read $Rd(c)$ and written $Wr(c)$ by a command c are defined as follows:

c	$Wr(c)$	$Rd(c)$
skip	$\emptyset$	$\emptyset$
$x := e$	x	$Vars(e)$
$c1; c2$	$Wr(c1) \cup Wr(c2)$	$Rd(c1) \cup Rd(c2)$
if b **then** $c1$ **else** $c2$	$Wr(c1) \cup Wr(c2)$	$Rd(c1) \cup Rd(c2) \cup Vars(b)$
while b **do** c	$Wr(c)$	$Rd(c) \cup Vars(b)$

where $Vars(e)$ is the set of all (read) variables in expression e and has a similar inductive definition.

1 A note on inductive proofs

So far we've seen two kinds of inductive proofs for program properties. One proof technique, structural induction, matches the syntactic structure of language constructs. We've also seen a non-inductive proof where we try to construct a proof tree in the conclusion using subtrees from the hypothesis.

But here is an example of a property where none of these techniques work: each IMP program c that terminates yields a unique final state. That is, if $\langle c, s\rangle \Downarrow s'$ and $\langle c, s\rangle \Downarrow s''$, then $s' = s''$.

Structural induction on c fails here (why? in which of the five structural cases?). What we need is induction on the derivation of $\langle c, s\rangle \Downarrow s'$.

Let $P(c, h) = \forall s, s', s'' : \langle c, s\rangle \Downarrow s'$ and $\langle c, s\rangle \Downarrow s''$ and $height(\langle c, s\rangle \Downarrow s') = h$ then $s' = s''$. We want to show $P(c, h)$ holds for all $c \in \mathsf{Com}$ and $h \geq 0$. Here, the height of the proof tree ignores the parts of the proof tree that refer to the evaluation of expressions; for instance the height of $\langle x := e, s\rangle \Downarrow s'$ is considered 0.

The proof by induction on derivations is essentially a proof by mathematical induction on h.

In the base case, $h = 0$. This corresponds to the evaluation of **skip** and of assignments $x := e$. We leave these cases as an exercise.

In the inductive case, assume $P(c, h)$ holds for some $h \geq 0$ and for all commands. We want to prove that $P(c, h + 1)$ also holds. Since $h + 1 > 0$, it means that the evaluation $\langle c, s\rangle \Downarrow s'$ has height at least 1, so c is either a sequence, an **if**, or a **while** command. We will analyze the case when c is a **while** loop (the other cases are left as an exercise).

Assume that s, s', s'' are stores such that $\langle \textbf{while } b \textbf{ do } c, s\rangle \Downarrow s'$ (with a height $h+1$ for the derivation), and $\langle \textbf{while } b \textbf{ do } c, s\rangle \Downarrow s''$. We want to show that $s' = s''$.

We have two subcases here, depending on whether the test condition b evaluates to **true** or **false**. If $\langle b, s\rangle \Downarrow \textbf{false}$, then, by inspecting the rule (fwhile), it must be the case that $s' = s$ and $s'' = s$. Therefore, $s' = s''$.

If $\langle b, s\rangle \Downarrow \textbf{true}$, then the rule (twhile) is applicable for both evaluations $\langle \textbf{while } b \textbf{ do } c, s\rangle \Downarrow s'$ and $\langle \textbf{while } b \textbf{ do } c, s\rangle \Downarrow s''$. We get:

$$\frac{\langle b, s\rangle \Downarrow \textbf{true} \quad \langle c, s\rangle \Downarrow s_0 \quad \langle \textbf{while } b \textbf{ do } c, s_0\rangle \Downarrow s'}{\langle \textbf{while } b \textbf{ do } c, s\rangle \Downarrow s'}$$

$$\frac{\langle b, s\rangle \Downarrow \textbf{true} \quad \langle c, s\rangle \Downarrow s_1 \quad \langle \textbf{while } b \textbf{ do } c, s_1\rangle \Downarrow s''}{\langle \textbf{while } b \textbf{ do } c, s\rangle \Downarrow s''}$$

The key fact here is that the evaluations of both c and **while** b **do** c in the premises have height h. Therefore, we can apply the induction hypothesis $P(c, h)$ for both of them, By induction hypothesis for c we get that $s_0 = s_1$. Then we apply the induction hypothesis for **while** b **do** c (since we know that $s_0 = s_1$) and get $s' = s''$, which concludes the proof.