Introduction to the theory of computation
week 5 (chapters 4 and 2 of Course Notes)

14th June 2005

Proving non-membership in recursively-defined sets

Occasionally you may want to prove that some element, or class of elements, cannot belong to a set. If the set has been defined by recursively, one way NOT to prove non-membership is to generate a list of set elements and demonstrate that the forbidden element (or class of elements) isn't/aren't on the list. The problem is that the list is probably infinitely long.

An approach that often works is to think of some property of a recursively-defined set that the element violates. If you can prove that every element $e \in E$ has a property, and that the element being discussed, x, does not, then $x \notin E$. Consider an example from our set E of well-formed arithmetic expressions.

Claim: If n is a natural number, then no element of E has $6n$ characters.

You can examine lots of elements of E and verify that they don't have a natural multiple of 6 characters, but this lack of examples isn't a proof. After trying a few examples, it becomes apparent that all the well-formed arithmetic expressions have an odd number of characters. If you could prove this, then this property would make it easy to prove that no natural multiple of 6 is in E.

Claim: Let $P(e)$ be "e has an odd number of characters." Then for all $e \in E$, $P(e)$.[1]

Now if x has $6n$ characters, $x \notin E$, since $P(x)$ is false.

Recursively defined sets as trees

It is sometimes useful to think of a recursively-defined set as a tree. The most basic elements are the leaves, and the internal nodes combine elements "below" them using the operators defined in the induction step. For example, the expression $((x+y)\times((y\div z)+x))$ as a tree looks like (well, see etree.pdf on the web page). Once you convince yourself there is a correspondence between the recursively-defined set and trees, consider

- What kind of trees are they?
- What does the claim (from last week) that $op(e)+1 = vr(e)$ emean if we consider these as trees?
- What connection can you make with this week's claim that expression in E have an odd number of characters and trees?

If you make a correspondence between recursively-defined sets and trees, then you can transfer things you know about trees to the recursively-defined sets, and vice versa. If you formulate the correspondence carefully, any fact that you prove about trees leads to a proof of the analogous fact about the recursively-defined set.

In a completely different direction, you can use the technique of recursive definition to define trees (see Course Notes), but we don't visit that idea during this course.

What do we mean by a correct program?

We now discuss a (probably under-used) concept incomputer science, program correctness. We limit our discussion to terminating, deterministic programs (the correctness of non-terminating daemons are beyond the scope of this course).

We want a program to "do what we intend" every time we run it. To make this desire precise, we say that a program is CORRECT with respect ot its specification if whenever the user satisfies the PRECONDITION, the program terminates and satisfies the POSTCONDITION.

PRECONDITION: The user guarantees the state of relevant program variables when the program begins.

POSTCONDITION: The author(s) of the program guarantee(s) it will terminate, and at that time the program variables will be in the specified state.

If the precondition is violated, all bets are off. The program could crash, toxic sludge could seep over the user's fingers, or the program variables could be in some unspecified state. If the precondition is met, the program author(s) guarantee that the postcondition specifies the state of program variables. Consider the following precondition/postcondition pair:

PRECONDITION: A is an integer array, x is an integer.

POSTCONDITION: Return i such that $A[i] == x$, or else $A.length$ if A has no element equal to x.

Some common-sense assumptions aren't stated. We assume (but don't state) in the precondition that there is a stable supply of electricity (or whatever fuel the computer runs on), and that the immediate neighbourhood isn't fully of baseball-bat-wielding computer smashers. We assume (but don't state) that the program doesn't change the contents of A (otherwise there is a trivial way to guarantee the postcondition — how?)[2]. When in doubt, state assumptions.

In what follows I will be using Java as a common language. Ask questions about anything that is unclear.

Binary search example

Here's a description of binary search in words:

"If you have a sorted list with more than one element, examine the middle element to decide whether you want to search the first half or the second half of the list. Repeat this halving until you have a list of one element."

This intuitive description needs the details to be firmed up. What is the "middle," what are the "first half" and the "second half"? Consider the computer listing for binary search (see web page).

The program seems plausible: f and l bracket the area to be searched, and they get closer together until they either converge on the value you want, or exclude the entire array as a legitimate area to be searched. But how do you know you should set l to m, and not, say, $m-1$? How about setting f to $m+1$ instead of m?

We break up correctness into two parts:

1. PARTIAL CORRECTNESS: If the precondition is satisfied and $binSearch$ terminates, then the postcondition is satisfied. In other words, if A is a sorted array of length at least 1, x is an integer, and $binsearch(A, x)$ terminates, then it either returns t such that $A(t) == x$, or else it returns $A.length$.

2. TERMINATION: If the precondition is satisfied, then $binSearch$ terminates. In other words, if A is a sorted array of length at least 1, then $binSearch(A, x)$ terminates.

Taken together, these would guarantee that *binSearch* is correct with respect to its specification. It is usually easier to prove the two parts of correctness separately. Here's how to go about it.

To prove partial correctness, you need to show that if the program terminates (when $f == l$), then $A(f) == x$ (so *binSearch* returns f), or else $x \notin A$ (so *binSearch* returns $A.length$). The informal reason we think this is true is that we think that f and l always bracket the area to be searched, so when $f == l$ the area to be searched consists of a single element.

This idea of bracketing can be made sharper if you think about what happens when there are two or more instances of x in the array. Which instance does *binSearch* return the index of?[3] We believe this bracketing property is INVARIANT throughout every iteration of the loop, so if we denote the value of variables at the end of the ith iteration of the loop by a subscript i, and if we denote the lowest index of x in the array (if it exists) as t_x, then we believe that the following loop invariants are true:[4] If we could prove these were true, then partial correctness won't be difficult.

Let $P(i)$ be "If the precondition of *binSearch* is satisfied and the loop as at least i iterations, then $0 \leq f_i \leq l_i \leq A.length - 1$ and $(t_x \in [f_i, l_i]) \vee x \notin A$, where t_x denotes the lowest index, if it exists, such that $A[t_x] == x$."

CLAIM: $P(i)$ is true for all $i \in \mathbf{N}$.[5]

Now combine the loop invariant with the assumption that *binSearch* terminates:

CLAIM: The precondition plus termination imply the postcondition.[6]

This proves partial correctness. To prove termination, you need to make precise your intuition that if the gap from f to l gets steadily smaller, but never less than zero, eventually you must have $f = l$. In symbols this says that if $g_i = l_i - f_i$, then g_i is always non-negative, and if g_{i+1} exists, then $g_i > g_{i+1}$. If you can prove these two things, then the sequence $\langle g_0, g_1, \ldots \rangle$ is finite (by well-ordering it has a smallest element g_k, and hence no g_{k+1}), so there are finitely many loop iterations. Since l_i and f_i are integers, so is there difference, and (by the loop invariant proved above) $l_i \geq f_i$, so g_i is a natural number. It remains to show that $\langle g_i \rangle$ is strictly decreasing.

CLAIM: If the loop is executed at least $i + 1$ times, then $g_{i+1} < g_i$.[7]

CLAIM: Suppose the precondition is satisfied. Then $binSearch(A, x)$ terminates.Proof: The sequence $\langle g_i \rangle$ is composed of natural numbers, since l_i and f_i are integers with $l_i \geq f_i$ by the loop invariant. The set of values in $\langle g_i \rangle$ form a non-empty subset of $\mathbf{N}$ (containing at least $l_0 - f_0$), and hence have a smallest element g_k. Since (by the previous claim) $\langle g_i \rangle$ is strictly decreasing, g_k is the last element, hence there are no more than k loop iterations and $binSearch(A, x)$ terminates. QED.[8]

Notes

[1]Proof (structural induction on e): If e is defined in the basis, then $e \in \{x, y, z\}$, so e has 1, hence an odd number, of characters. Thus $P(e)$ holds for the basis.

Induction step: Assume that $P(e_1)$ and $P(e_2)$ hold for arbitrary elements $e_{1,}e_{2,} \in E$, and that $e = (e_1 \oplus e_2)$, where $\oplus \in \{+, -, \times \div\}$. I must show that this implies $P(e)$. Expression e has the same characters as e_1 and e_2, plus an extra opening parenthesis, closing parenthesis, and operator $\oplus$ — three extra characters in all. By the IH e_1 has an odd number of characters, say $2k + 1$, for some integer k, and e_2 has an odd number of characters, say $2j + 1$, for some integer j. This means that e has $3 + 2j + 1 + 2k + 1 = 2(j + k + 2) + 1$ characters, an odd number since $j + k + 2$ is an integer (integers are closed under addition). Thus $P(\{e_1, e_2\}) \Rightarrow P(e)$, an arbitrary element of E defined in the induction step.
I conclude that $P(e)$ is true for all $e \in E$.

[2]Set $A[0] = x$ and return 0.

[3]the first one.

[4]

$$0 \leq f_i \leq l_i \leq A.length - 1$$
$$(t_x \in [f_i, l_i]) \vee x \notin A$$

[5]Proof (induction on i): $P(0)$ states that if the precondition of *binSearch* is satisfied, and the loop has at least i iterations, then $0 \leq f_0 \leq l_0 \leq A.length - 1$ and $(t_x \in [f_i, l_i]) \vee x \notin A$. Inspecting the program we see that $f_0 = 0$ and $l_0 = A.length - 1$, so the first part of the invariant is true, and either $t_x \in [0, A.length - 1$ or else $x \notin A$, so the claim holds for the base case.

Induction step: Assume that $P(i)$ holds for some arbitrary natural number i. If there is no $(i + 1)th$ iteration, then $P(i + 1)$ holds vacuously (empty antecedent). Otherwise, $f_i \neq l_i$, so (since $f_i \leq l_i$) we must have $f_i < l_i$. This means that

$$\begin{aligned} m_{i+1} &= (f_i + l_i)/2 \\ \text{integer division monotonic :} \quad &\geq (f_i + f_i)/2 \\ &= f_i \end{aligned}$$

... and you also

$$\begin{aligned} m_{i+1} &= (f_i + l_i)/2 \\ &\leq (l_i - 1 + l_i)/2 \\ \text{integer division floors real division} \quad &= \lfloor (l_i - 1 + l_i)/2.0 \rfloor \\ &= \lfloor l_i - \frac{1}{2} \rfloor \\ &< l_i \end{aligned}$$

So $f_i \leq m_{i+1} < l_i$, and we need to consider two cases.

1. If $A[m_{i+1}] \geq x$, then you set $f_{i+1} = f_i \leq m_{i+1} = l_{i+1}$, and so $0 \leq f_{i+1} \leq l_{i+1} \leq A.length - 1$, as wanted.
2. If $A[m_{i+1}] < x$, then you set $f_i < f_{i+1} = m_{i+1} + 1 \leq l_{i+1} = l_i$, so $0 \leq f_{i+1} \leq l_{i+1} \leq A.length - 1$, as wanted.

In both cases, the first invariant holds. For the second invariant, consider two cases

1. If $A[m_{i+1}] \geq x$, then if t_x exists you must have $t_x \leq m_{i+1} = l_{i+1}$ (since A is sorted), and since $f_i = f_{i+1}$, by the IH $t_x \geq f_{i+1}$, so we have $t_x \in [f_{i+1}, l_{i+1}]$, or else $x \notin A$.
2. If $A[m_{i+1}] < x$, then if t_x exists we must have $t_x \geq m_{i+1} + 1 = f_{i+1}$ (since A is sorted), and since $l_{i+1} = l_i$, by the IH $t_x \leq l_{i+1}$, so either $t_x \in [f_{i+1}, l_{i+1}]$ or $x \notin A$.

In either case the second invariant holds. We have shown that $P(i) \Rightarrow P(i+1)$, so we conclude that $P(i)$ holds for all $i \in \mathbf{N}$. QED.

[6]Proof: Suppose *binSearch* terminates at the end of the kth loop iteration. Examination of the loop condition implies that $f_k = l_k$. The loop invariant, $P(k)$, implies that either $t_x \in [f_k, f_k]$, in which case *binSearch* returns $t_x = f_k$, and $A[t_x] = x$, or else $x \notin A$, and (since $0 \leq f_k \leq A.length - 1$ implies that f_k is a valid index for A), $A[f_k] \neq x$, so $binSearch(A, x)$ returns $A.length$. In either case $binSearch(A, x)$ satisfies the postcondition, as claimed. QED.

[7]Proof: Suppose the loop iterates at least $i + 1$ times. Since it doesn't terminate at the end of loop i, we must have $f_i < l_i$, so (by result in loop invariant) $f_i \leq m_{i+1} < l_i$. If $A[m_{i+1}] \geq x$, then (by the program) $f_{i+1} = f_i$ and $l_{i+1} = m_{i+1}$, and so $g_{i+1} = l_{i+1} - f_{i+1} = m_{i+1} - f_i < l_i - f_i = g_i$, and the claim holds. If $A[m_{i+1}] < x$, then (by the program) $f_{i+1} = m_{i+1} + 1$ and $l_{i+1} = l_i$, so $g_{i+1} = l_{i+1} - f_{i+1} = l_i - m_{i+1} - 1 < l_i - f_i = g_i$, and the claim holds. In both cases the claim holds. QED.

[8]Proof: The sequence $\langle g_i \rangle$ is composed of natural numbers, since l_i and f_i are integers with $l_i \geq f_i$ by the loop invariant. The set of values in $\langle g_i \rangle$ form a non-empty subset of $\mathbf{N}$ (containing at least $l_0 - f_0$), and hence have a smallest element g_k. Since (by the previous claim) $\langle g_i \rangle$ is strictly decreasing, g_k is the last element, hence there are no more than k loop iterations and $binSearch(A, x)$ terminates. QED.