**Proofs of Program Correctness**

**Establishing Program Correctness**

Today's dominant practice in the software industry (and when writing up assignments) is to prove program correctness empirically. The simplest form of this technique consists of feeding various inputs to the tested program and verifying the correctness of the output. In some cases exhaustive testing is possible, but often it is not.

More sophisticated versions of this technique try to choose the inputs so that all, or at least the majority of the possible execution paths are examined. Independent of their degree of sophistication, these empirical methods do not actually that the respective program is correct.

The only thing we can actually prove with the empirical approach is that the program is not correct - for this purpose, a single example of incorrect behavior suffices. Absent an observation of incorrect behavior, however, we can not know - in general - whether the program is correct, or whether we have just not tested it with an input that would trigger an error.

As we all know, incorrect program behavior is pervasive. Some program errors are only irritating, but some can endanger life and limb. Next time you fly spend a minute pondering the importance of correct program behavior for your airplane's navigational system. Would you like the manufacturer to "think" that the program is correct based on a number of empirical tests, or would you prefer an unambiguous and definitive proof?

Once we establish the importance of program correctness as both an engineering and theoretical problem, we can turn our attention to actually solving it. Unfortunately, we can't do too much. It can be proven, for example, that there are no general algorithms that check (or prove) that a program correctly implements a given specification. We can not give general solutions for much simpler problems either. For example, there is no algorithm that would be able to decide whether an arbitrary program will terminate execution for a given input. Wouldn't it be interesting to have a tool that would prevent a program from even starting because the tool could determine a priori that the program would get into an infinite loop?

Since we can solve almost no general problem in the area of program correctness we must set more modest goals. Luckily, the lack of general solutions does not mean that program correctness can not be proven in certain particular cases, or in a context that is restricted in some sense. Today we are going to discuss two program correctness proofs that use the substitution model and induction.

**Induction Proofs**

Induction is a technique we can use to prove that certain properties hold for each element of a suitably chosen infinite set. The most common form of induction is that of mathematical induction, whereby we establish the truth of statement for all natural numbers, or - more generally - for all elements of a sequence of numbers. Induction can also be performed on more complicated sets, like pairs of non-negative integers, or binary trees (see below).

An inductive argument can be thought of as being not a proof per se, but a recipe for generating proofs. First, the relevant property P(n) is proven for the base case, which often corresponds to n = 0 or n = 1. Then we assume that P(n) is true, and we prove P(n+1). The proof for the base case(s) and the proof that allows us to go from P(n) to P(n+1) provide a method to prove the property for any given m >= 0 by successively proving P(0), P(1), ..., P(m). We can't actually perform the infinity of proves necessary for all choices of m >= 0, but the recipe that we provided assures us that such a proof exists for all choices of m.

To reduce the possibility of error, we will structure all our induction proofs rigidly, always highlighting the following four parts:

1. The general statement of what we want to prove;
2. The specification of the set we will perform induction on;
3. The statement and proof of the base case(s);
4. The statement of the induction hypothesis (generally, we will assume that P(n) holds, but sometimes we need stronger assumptions, see below), the statement of P(n+1) and proof of the induction step (or case).

We prefer that you use precise mathematical notation for all the statements that you make (see our examples below). If you are not familiar enough with mathematical notation, however, we also accept semi-informal statements in plain English, assuming that they are correct, unambiguous, and complete. For example, the statement "given a natural number n >= 0, the sum of the first n natural numbers is equal to n(n + 1) / 2" is acceptable. However, the statement "the sum of the first n numbers is n(n + 1) / 2" is incomplete: we are not given enough information about n. Of course, we can infer some things about n - it can not be a general real number, or a negative number, because then the phrase "the first n numbers" (by the way, what kind of numbers?) would not make sense. Don't let us guess, however, fill in all the necessary details!

As mentioned in passing above, induction is feasible on sets that are more general than the set of natural numbers. Technically, induction can be performed on all well-founded sets.

**Aside**

A well-founded set is a set endowed with a partial order such that the set contains no infinite descending chains. Equivalently, and perhaps more intuitively, a well-founded set is a partially ordered set in which every non-empty subset has a minimal element.

A partial order on a set S is a reflexive, transitive, and antisymmetric binary relation on S x S (the Cartesian product of S with itself).The binary relation is often denoted by <=, and one can think of it as a generalization of the "less than or equal to" comparison operator to elements of S.

A minimal element m of U in S is an element of U such that no other element u of U has the property that u <= m. It is possible for a subset of a partially ordered set to have more than one minimal element.

Fix an element s(0) of S, and consider elements s(1), s(2), ..., s(n) in S, such that s(n) <= ... <= s(2) <= s(1) <= s(0). We say that (s(0), s(1), s(2), ..., s(n)) forms a descending chain in S. Note that s(0), s(1), s(2), ... are all "smaller" than any of the elements that precede them in the chain. The property of having no infinite descending chains is equivalent to saying that no matter what our choice for s(0) is, and no matter how we choose s(1), s(2), ..., s(n), at some point we will not be able to extend the descending chain further by finding a suitable element of S.

Intuitively, following a descending chain corresponds to reducing the problem of proving P(s(i)) to proving P(s(i+1)) (do not get confused by the increasing index, s(i+1) <= s(i)). The fact that all descending chains are finite guarantees that sooner or later we will reach a problem that can not be reduced further, a base case. The minimal elements of S form the base cases to be considered in the induction proof.

Consider, for example, the set of pairs of natural numbers (i.e. S = N x N) endowed with the following comparison operation: (p, q) <= (r, s) iff (p < r) or (p = r and q <= r). With this definition, any two pairs (p, q) and (r, s) can be compared (i.e. at least one of (p, q) <= (r, s) and (r, s) <= (p, q) holds). Such a set is a totally ordered set; totally ordered set are also partially ordered sets. Quite clearly, (0, 0) is the smallest element of this set.

Let us change the definition above slightly. Consider S = N x N, as before but define (p, q) <= (r, s) iff (p <= r) and (q <= s). This set is partially ordered, but not totally ordered; for example, neither (5, 7) <= (6, 4), nor (6, 4) <= (5, 7) holds. Given set X = {(5, 7), (6, 4), (10, 10)}, its minimal elements are (5, 7) and (6, 4). Can you say which are the minimal elements of S?

**End Aside**

Sometimes we can not prove P(n+1) based solely on P(n), but we can succeed with the proof if we assume that the property holds for all natural numbers m <= n. Formally: given a natural number n >= 0 we want to prove that (for all m s.t. 0 <= m <= n: P(m) holds) => P(m+1) holds. This form of induction is called strong induction.

Note: there exist still other forms of induction; a particularly interesting example is that of structural induction, which you will discuss in section.

**The Substitution Model and Correctness Proofs**

**Induction on the Set of Natural Numbers**

Consider the well-known example of the factorial function:

fun fact n =

if n = 0

then 1

else n \* fact (n - 1)

We want to prove formally that fact(n) = n! for all natural numbers n. We write down the proof by following precisely the four steps specified above:

1. We want to prove that, for all natural numbers n, fact(n) = n!
2. We perform induction on the set of natural numbers N.
3. Base case (corresponds to n = 0): we need to prove that fact(0) = 0! = 1.

We perform the proof by using the substitution model. To reduce the size of the proof we will skip all non-essential steps in the application of the substitution model. Further, we introduce the notation BODY = if n = 0 then 1 else n \* fact (n - 1).

In keeping with our previous discussion, we should imagine that there is a let statement wrapping the definition of fact, and that fact(0) is the part of the let between in and end. We start applying the substitution model after the function expression defining fact has already been substituted in fact(0). We will do this in other proofs as well, primarily to reduce the length of the proof.

eval(fact(0))

=eval((fun fact n = BODY) 0)

= eval((fn n => if n = 0 then 1 else n \* (fun fact n = BODY) (n - 1)) 0)

= eval(if 0 = 0 then 1 else 0 \* (fun fact n = BODY) (0 - 1))

= 1 [because the condition of the if statement is true

we evaluated the 'then' branch]

This proves the base case.

1. Assuming that, for any given natural number n, fact n = n!, we will prove that fact (n + 1) = (n + 1)!

Note: here n + 1 is not an SML expression but a notation for the successor of n in the set of natural numbers. Because n + 1 is not an expression, you should not not try to evaluate it in the context of the substitution model. If this confuses you, you can introduce the notation m = n + 1, and just use m whenever we would have used n + 1.

We should point out that you should not confuse the two uses of n: we used the same symbol for the formal argument of function fact, and for the natural number that appears in the induction hypothesis. If you are confused by this, you can rewrite the definition of fact to use another identifier for its argument, say, k.

eval((fun fact n = BODY) (n + 1))

= eval((fn n => if n = 0 then 1 else n \* (fun fact n = BODY) (n - 1)) 0)

= eval(if (n + 1) = 0 then 1 else (n + 1) \* (fun fact n = BODY) ((n + 1) - 1))

= eval((n + 1) \* (fun fact n = BODY) ((n + 1) - 1))

[n >= 0 is an arbitrary natural number, hence (n + 1) > 0,

and this implies that the condition in 'if' evaluates to false]

Before we can evaluate the expression above we need to evaluate (fun fact n = BODY) ((n + 1) - 1). By taking a few shortcuts in the substitution model, we establish that this expression represents in fact fact(n), which we know from the induction hypothesis to be equal to n!.

eval((n + 1) \* (fun fact n = BODY) ((n + 1) - 1))

= [using the induction hypothesis]

eval((n + 1) \* n!)\

[ both n + 1 and n! are numbers (values) here, not SML expressions! ]

= (n + 1)!

This proves the induction step and completes the induction proof.

**Induction on the Length of Lists**

Consider the function definition below:

fun mapsquares (x: int list): int list =

if null(x) then []

else (hd x \* hd x)::mapsquares(tl x)

We want to prove that given a list l1 of integers in the input, this function returns a list of integers l2 such that the ith element of l2 is the ith element of l1 squared. To simplify our proofs, we will assume that functions null, hd, and tl are (or act like) predefined unary operators.

One might have considered writing mapsquares using a case expression. The style guide actually advises you against that, since a case with two alternatives is really equivalent to an if expression. Even if we ignore the style guide, we have not defined the substitution model for case expressions, so we could not use the alternative form of mapsquares in our proof.

The description above, while reasonable, allows for some elements of ambiguity. For example, we have not indicated explicitly that the length of the output list is identical to the length of the input list. To avoid these problems, we will use a more formal notation in our proof.

Operator #(i + 1) does not exist in SML. We use this form here as a notational convenience. The same observation applies to ^, used here to indicate exponentiation.

Here are the four parts of the induction proof:

1. P(n) := Given an arbitrary list x of n integers, mapsquares(x) returns a list z s.t. for all 1 <= i <= n we have (#i x) ^ 2 = (#i z). Note: when we say "f(a) returns b" we mean eval(f(a)) = b.
2. We perform induction on the length of lists, which is in the set of natural numbers N = {0, 1, 2, 3, ...}
3. The base case corresponds to n = 0, i.e. lists of 0 length. We instantiate P(n) to get P(0) as follows:

P(0) := Given an arbitrary list x of length n, mapsquares(x) returns a list z s.t. for all 1 <= i <= 0: (#i x) ^ 2 = (#i z).

Since the only list of length 0 is [], and no integer i satisfies 1 <= i <= 0, we can rewrite P(0) as follows:

P(0) := mapsquares([]) returns []

In other words: P(0) := eval(mapsquares([])) = [].

eval(mapsquares([]))

[substitute function body]

= eval((fun mapsquares (x) = if null(x) then [] else (hd x \* hd x)::mapsquares(tl x)) [])

[evaluate and substitute argument in function body]

= eval(if null([]) then [] else (hd [] \* hd [])::mapsquares(tl []))

|----

| [evaluate test condition]

|

| eval(null([])) = true by definition of the null operator

|

| [condition is true => evaluate 'then' branch]

|----

= eval([]) = []

[an empty list evaluates to itself;

think of it, not quite correctly, as a constant]

Thus we have determined that eval(mapsquares([])) = [], which proves the base case P(0).

1. Induction step: Assume that P(n) holds for an arbitrary natural number n. We will prove P(n + 1).

P(n + 1) : = Given an arbitrary list x of length n+1, mapsquares(x) returns a list z s.t. for all 1 <= i <= n+1 we have (#i x) ^ 2 = (#i z).

An arbitrary list of length n+1 can be obtained, for example, by prepending an arbitrary element of the base type to an arbitrary list of n elements. Thus if y is an arbitrary list of integers having length n, x = a::y (a integer) is an arbitrary list of length n+1. Note that since y has length n, the induction hypothesis applies to it.

eval(mapsquares(a::y))

[function body substitution]

= eval((fun mapsquares (x) = if null(x) then [] else (hd x \* hd x)::mapsquares(tl x)) [a::y])

[evaluate and substitute argument in function body]

= eval(if null(a::x) then a::x else (hd (a::x) \* hd (a::x))::mapsquares(tl (a::x)))

|-----

| [evaluate test condition]

|

| eval(null(a::y)) = false

|

| [list a::y contains at least one element (specifically, a),

| and thus it can't be an empty list]

|

|-----

[condition is false => evaluate 'else' branch]

= eval((hd (a::y) \* hd (a::y))::mapsquares(tl (a::y)))

[evaluate hd and tl applications, treat them as unary operators;

note that we are taking shortcuts!]

= eval((a \* a)::mapsquares(y))

As discussed above, the induction hypothesis applies to y, as y is a list of length n. We have the following relations:

If mapsquares(y) = z => (by I.H.) for all 1 <= i <= n property (#i y) ^ 2 = (#i z) holds.

Since (#i y) = (#(i+1) x) (i.e. y is the tail of x), then for all 1 <= i <= n we have that (#i y) ^ 2 = (#(i + 1) x) ^ 2 = (#i z). In other words, mapsquares(y) returns a list containing the squares of the last n elements of x, in their original order.

Now, a \* a is the square of the first element of x, i.e. (a \* a) = (#1 x) ^ 2.

We can now refer to the concatenation (a \* a)::mapsquares(y) = z', and state that (#1 x) ^ 2 = (#1 z'), and that for all 1 <= i <= n we have (#(i + 1) z') = (#i z) = (#(i + 1) x). By rewriting these expressions we get:

(#1 x) ^ 2 = (#1 z'), and that for all 2 <= i <= n + 1 we have (#(i z') = (#i x),

which is equivalent to

for all 1 <= i <= n + 1: (#i x) ^ 2 = (#i z'),

which proves P(n + 1).