Program Verification

In the late 1960s, research on program verification began.

- A 1968 NATO Summer Conference resulted in the term "software crisis"
- Idea was to improve software reliability throught automatic or semi-automatic means.

The result was a large number of automatic theorem provers:

1. The Boyer-Moore prover (from Texas), originally designed for veriying Lisp program.
2. The Larch Prover (from MIT), based on term-rewriting and equational logic. originally design to check formal specifications.
3. The Otter system (from Argonne National Labs) is a resolution-based theorem program. Widely used systems today.
4. NUPRL (from Cornell) is a proof system for an intuitionistic type theory based on Martin Lof type theory, with proof by refinement and extraction of programs from proofs.

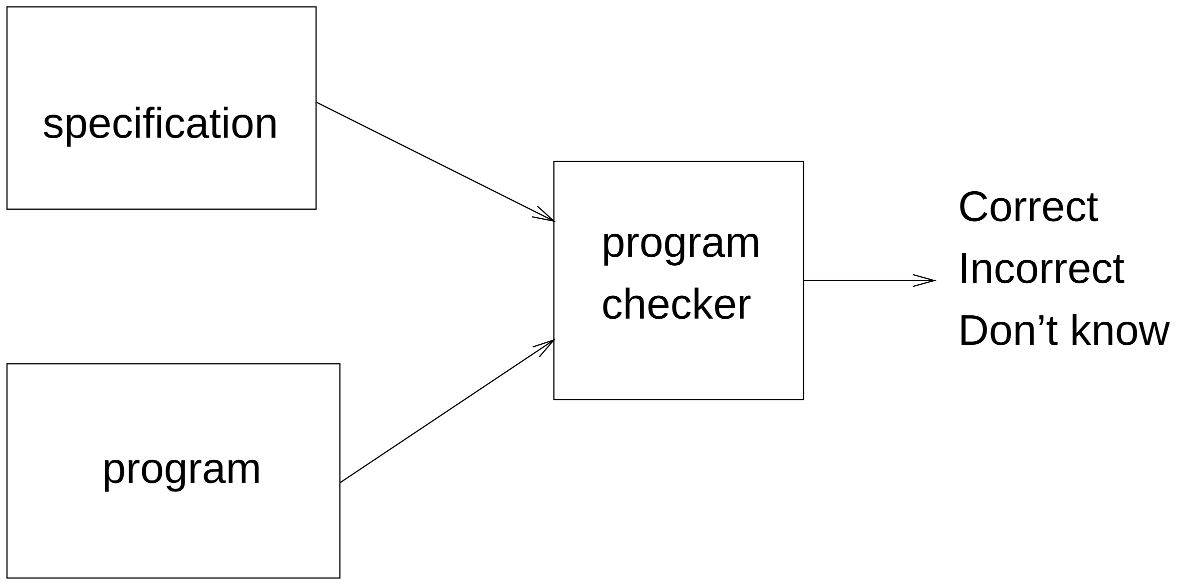
Status of Automatic Theorem Proving

The problem turned out to be much harder than originally hoped, and as a result, many U.S. funding agencies have abandoned it.

There are some positive results, however:

1. Hardware verification
2. Correctness proofs of architectures
3. Automatic proofs of distributed algorithms
4. In general, continuing interest and support in the European community.

The Program Verification Problem



There are at least three different problems that arise in automatic program verification:

1. writing formal specifications to describe the desired behavior
2. writing programs that are easier to verify
3. reasoning about the program itself

Program Correctness

What does is mean for a procedure to be a correct implementation of a specification? There are two standard notions of correctness:

- Partial correctness: if the precondition is true when the procedure is invoked, then if the procedure terminates, the postcondition must be true when the procedure returns.
- Total correctness: if the precondition is true when the procedure is invoked, then the procedure must terminate, and the postcondition must be true when the procedure returns.

```
{ P }
program text
{ Q }
```

This says that if `P is true before executing the given program text, then Q will be true afterwards.`

Proof Rules

With this notation, we can write inference rules such as the following.

$$\frac{P \supset P', \quad \{P'\}\ S\ \{Q'\}, \quad Q' \supset Q}{\{P\}\ S\ \{Q\}} \qquad \mathit{Consequence}$$

$$\frac{\{P\}\ S_1\ \{R\}, \quad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1; S_2\ \{Q\}} \qquad \mathit{Sequence}$$

$$\frac{\{P\ \wedge\ C\}\ S_1\ \{Q\}, \quad \{P\ \wedge\ \neg C\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if C then S}_1\ \texttt{else S}_2\ \texttt{fi}\ \{\texttt{Q}\}} \qquad \mathit{Conditional}$$

Note some hidden assumptions.

Proof Rule for Assignment

A plausible but invalid axiom is:

$$\{P\}\ \texttt{x := E}\ \{[\texttt{x}/\texttt{E}]P\}$$

Assume that P is $y \neq z$, so the assertion after executing `x := y` would be $x \neq z$.

$$\{y \neq z\}\ \texttt{x := y}\ \{x \neq z\}$$

However, if x is actually an alias for z, then this is false.

The general problem is that substitution does not preserve validity.

The valid proof rule for assignment will instead result in assertions being pushed backwards through the program.

$$\{[\texttt{E}/\texttt{x}]P\}\ \texttt{x := E}\ \{P\}$$

Formal Verification of Straight-Line Code

A very simple function, written with extra variables to make the proof easier.

```
// Postcondition: returns x^3
int cube(int x) {
  int res;

  res = x*x;

  res = res*x;
  /* assert(res = x*x*x) */
  return res;
}
```

Proof Idea: walk backwards through the program, adjusting the assertion for each assignment statement.

Proving Properties of Loops

There are two techniques for proving things about loops, one for partial correctness and one for termination.

- *loop invariant*: a predicate that is true before and after each iteration of the loop body.
- *variant function*: an integer valued function that gets closer to zero on every iteration of the loop, and once the function is less than or equal to zero, the loop will terminate.

Note: variant functions describe here are a special case. In general and value domain with a Noetherian order will do (no infinite decreasing chains). E.g., lexicographic orderings often work.

Concentrate on partial correctness for now.

Partial Correctness

Consider the problem of reasoning about a program with a loop.

```
/* assert(P) */
while (B) { S; }
/* assert(Q) */
```

Informally, the invariant I must satisfy the following properties:

1. I must be true before we execute the while loop. I.e., $P \Rightarrow I$.
2. If I is true before evaluating B, it is true after evaluating B. (This is easy to gaurantee if B does not contain any assignment statements, which is generally good programming style, anyway.)
3. If I and B are true at the beginning of the loop body, then I must be true at the end of the loop
4. I plus the negation of the condition (i.e., $I \wedge \neg B$) must imply the postcondition Q.

Proof Rule for Loops

The proof rule for while loops makes use of such an invariant property.

$$\frac{\{I \wedge B\}\ \texttt{S}\ \ \{I\}}{\{I\}\ \texttt{while B do S od}\ \{\texttt{I} \wedge \neg\texttt{B}\}}\ loops$$

Ignoring the non-local jumps outside of loops and side effects in B.

Note: To reason about `for` loops, first transform:

```
for (S1; B; S2) {        into        S1; while (B) {
   S;                                               S;  S2;
}                                                   }
```

Example: Linear Search

```
// Precondition:  n > 0 and the size of a == n.
// Postcondition: Returns res.  If key is in a,
//            a[res] = key, otherwise res = -1.
 int search (int a[], int n, int key) {
   int res = -1;
   int i = 0;
   while (i < n) {
     if (a[i] == key) {
          res = i;
        }
     else i++;
   }
   return res;
 }
```

A loop invariant is:

(res = -1) => (forall j, 0 <= j < i, a[j] != key)

$\wedge$ (res != -1) => a[res] = key.

Annotated Program

```
res = -1; i = 0;
/* assert (n>0 & size of a == n) */
  while (i < n) {
    /* assert((i < n)
       & (res = -1) => (forall j, 0<=j<i, a[j] != key));
       & (res != -1) => a[res] = key */
    if (a[i] == key) res = i;
    /* assert((i < n)
       & (res = -1) => (forall j, 0<=j<i+1, a[j] != key));
       & (res != -1) => a[res] = key */
    else i = i+1;
    /* assert((i < n)
       & (res = -1) => (forall j, 0<=j<i, a[j] != key));
       & (res != -1) => a[res] = key */
  } /* assert((i >= n)
       & (res = -1) => (forall j, 0<=j<n, a[j] != key));
       & (res != -1) => a[res] = key */
return res;
```

Reasoning about ADTs

Verifying and ADT is essentially a set of function verifications, one for each member function.

Some functions may depend on a representation invariant for their correctness. We therefore add to the proof rules:

- For each constructor, prove that the representation invariant holds after construction, and for each each non-constructor, prove the representation holds after the function completes.
- For each non-constructor member function, assume the representation invariant holds on each object in the class when the function is called. (This includes the argument passed into the copy constructor.)

We still have the problem of notation for ADTs and the meaning of that notation.

Informal Specifications of an ADT

```
class set {
  set();
   // post: constructs a
   //  new empty set
  insert(int i);
   // post: insert i
  int choose()     ;
   // pre: the set is not empty
   // post: chooses some element,
   //  deletes it and returns it
  bool isin (int i)        ;
   // post: if i is in the set
   //  returns true, else false
    };
```

Regardless of level of formality, we can ask certain questions about the interface and about its specification:

1) can it be implemented?

2) is the interface adequate?

construct every value

observe distinctions between different values

3) is it deterministic?

Formal Specifications of an ADT

```
class set {
  set();
   // post: this = {};
  insert(int i);
   // post: this' = this ∪ {i};
  int choose();
   // pre: this != {};
   // post: ∃ i, i ∈ this & i ∉ this'
      this = this ∪ {i} & choose = i
  bool isin (int i)
   // post: isin = (i∈ this)
```

Questions about the formal specification:

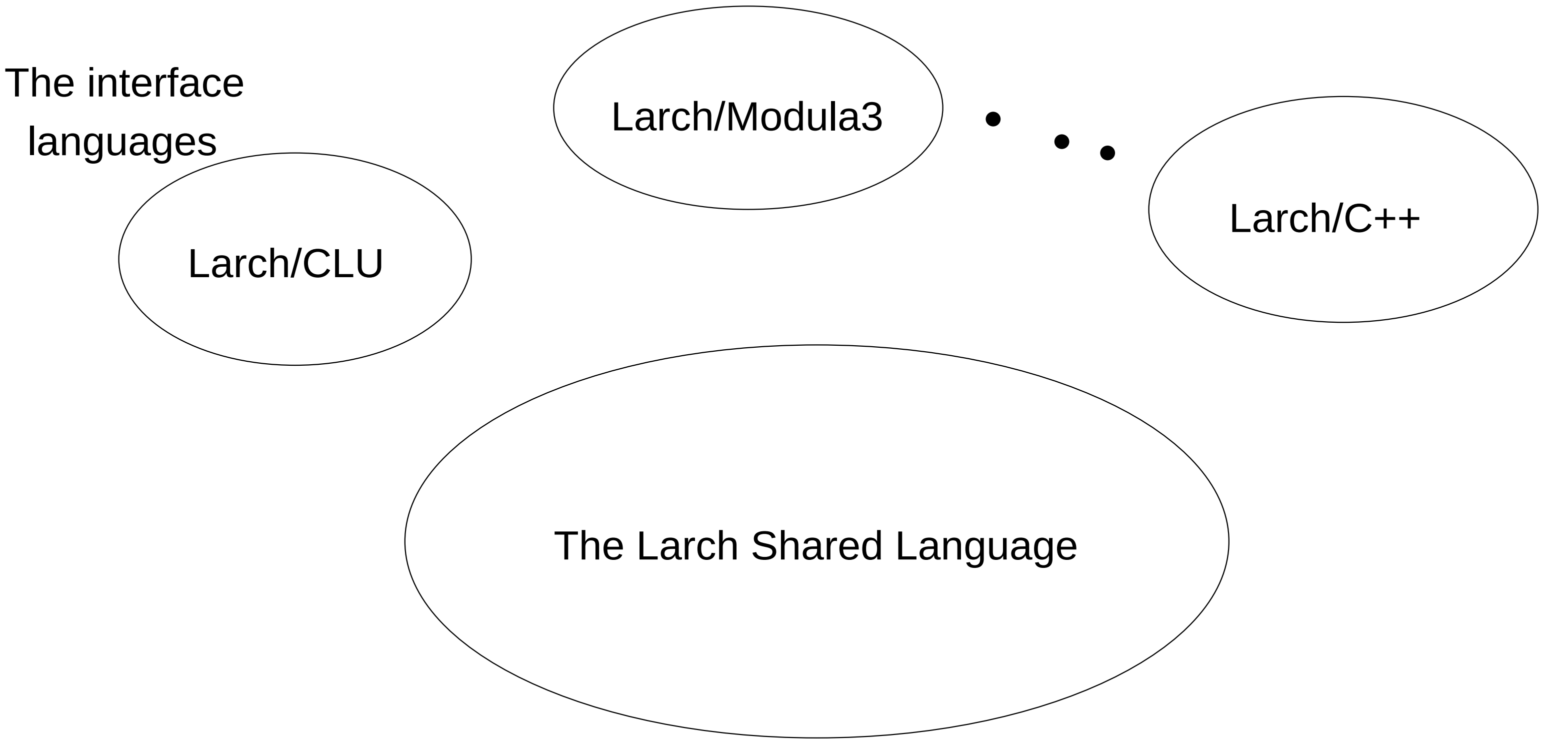
1) is it consistent?

2) is it trivial?

3) is it sufficient?

This wasn't too hard to write for sets, but what about other algorithms or data structures? Where will the notation come from and what axioms does it have?

Larch Family of Specification Languages



Define names for (mathematical) values, along with axioms that describe properties of the values.

```
String_Table      trait
  introduces
    new: -> String_Table
    add: String_Table, String, Int -> String_Table
    delete: String_Table, String -> String_Table
    is_in: String_Table, String -> Bool
    eval: String_Table, String -> Int
    size: String_Table -> Int

  constrains   new, add, delete, is_int, eval, size
     so that for all   [s, s1: String, v: int, t: String_Table]

        <axioms go here>
```

constrains new, add, delete, is_int, eval, size

so that for all [s, s1: String, v: int, t: String_Table]

1) eval(add(t, s, v), s1) = if s = 1 then v else eval(t, s1)

2) isin(s, new) = false

3) isin(s, add(t,s1,v)) = ((s = s1) | isin(s,t))

4) size(new) = 0

5) size(add(t,s,v)) = if isin(s,t) then size(t) else size(t)+1

6) delete(new,s) = new

7) delete(add(t,s,v), s1) = if s=s1 then delete(t,s1) else add(delete(t,s1),s,v)

Theory of Traits

The set of theorems provable from axioms is called a theory.

The theory associated with a trait is made up of the following wff:

1. all terms fo sort Bool
2. all equations t1 = t2, where t1 and t2 are terms of the same sort
3. quantified wff forall and there exists

It contains the equations given in the "contrains" clause, plus wff inferred by:

1. reflexivity	t = t
2. symmetry	t = s => t = s
3. transitivity	t1 = t2 & t2 = t3 => t1 = t3
4. conguence	t1=t1’ & ... & tn=tn’ & f(t1,...,tn)=f(t1’,...,tn’) forall f
5. instantiation	t1 = t2 => [x<-t](t1) = [x<-t](t2)

The equational theory
plus theory of booleans and first order predicate calculus rules.

Example Proof

size(add(add(new, x, y), x, z)) =

if isin(x, add(new,x,y)) then size(add(new,x,y)) else 1+size(add(new,x,y))

if (x=x) | isin(x,new) then size(add(new,x,y)) else 1+size(add(new,x,y))

if true then size(add(new,x,y)) else 1+size(add(new,x,y))

size(add(new,x,y))

if isin(x,new) then size(new) else 1+size(new)

if false then size(new) else 1+size(new)

1+size(new)

1+0 = 1

not in the theory so far....

Many proofs are done by structural induction on terms e.g., ∀ t:String_Table . size(t)>=0