Body of Split

remember
assume $r = \Lambda \wedge c = \Lambda$
Test_If_Empty(q, q_is_empty)
loop
  maintaining $(q \neq \Lambda \Rightarrow |r| = |c|) \wedge |r| \leq |c| \wedge$
    $|c| \leq |r| + 1 \wedge$
    $\text{IS\_PERMUT}(c * q * r, \#c * \#q * \#r) \wedge \ \ldots$
  while not q_is_empty **do**
    Dequeue(q, x)
    Enqueue(c, x)
    Test_If_Empty(q, q_is_empty)
    if not q_is_empty **then**
      Dequeue(q, x)
      Enqueue(r, x)
    end if
    Test_If_Empty(q, q_is_empty)
end loop
confirm $|r| \leq |c| \wedge |c| \leq |r| + 1 \wedge$
    $\text{IS\_PERMUT}(c * r, \#q)$

Additional Conjunct for Invariant

$$\ldots \wedge (\text{q\_is\_empty} \Leftrightarrow \text{q} = \Lambda)$$

```
– – (0)
  assume r = Λ ∧ c = Λ
  Test_If_Empty(q, q_is_empty)
– – (1)
  loop
    maintaining (q ≠ Λ ⇒ |r| = |c|) ∧ |r| ≤ |c| ∧  . . .
    while not q_is_empty do
– – (2)
      Dequeue(q, x)
– – (3)
      Enqueue(c, x)
– – (4)
      Test_If_Empty(q, q_is_empty)
– – (5)
      if not q_is_empty then
– – (6)
        Dequeue(q, x)
– – (7)
        Enqueue(r, x)
– – (8)
      end if
```

– – (9)

Test_If_Empty(q, q_is_empty)

– – (10)

end loop

– – (11)

confirm $|\mathrm{r}| \leq |\mathrm{c}| \wedge |\mathrm{c}| \leq |\mathrm{r}| + 1 \wedge \ \ldots$

Queue Operations

procedure Test_If_Empty (q: Queue
empty: Boolean)
ensures "$(\#q = q) \wedge (q = \Lambda \Leftrightarrow empty)$"
procedure Dequeue (q: Queue
x: Item)
requires "$q \neq \Lambda$"
ensures "$\#q = \langle x \rangle * q$"
procedure Enqueue (q: Queue
x: Item)
ensures "$q = \#q * \langle \#x \rangle \wedge$ **is_initial**(x)"

Example Facts

1.1 $(\text{q\_is\_empty}_1 \Leftrightarrow \text{q}_1 = \Lambda) \wedge \ \ldots$

2.1 $\neg\text{q\_is\_empty}_1 \Rightarrow$
$(\neg\text{q\_is\_empty}_2 \wedge \ \ldots$
$\wedge (\text{q\_is\_empty}_2 \Leftrightarrow \text{q}_2 = \Lambda))$

3.1 $\neg\text{q\_is\_empty}_1 \Rightarrow$
$(\langle \text{x}_3 \rangle * \text{q}_3 = \text{q}_2 \wedge \text{r}_3 = \text{r}_2 \wedge \text{c}_3 = \text{c}_2)$

4.1 $\neg\text{q\_is\_empty}_1 \Rightarrow$
$(\text{c}_4 = \text{c}_3 * \langle \text{x}_3 \rangle \wedge \text{r}_4 = \text{r}_3 \wedge \text{q}_4 = \text{q}_3)$

9.1 $\neg\text{q\_is\_empty}_1 \Rightarrow (\ \ldots \wedge$
$(\text{q\_is\_empty}_5 \Rightarrow$
$(\text{x}_9 = \text{x}_5 \wedge \text{c}_9 = \text{c}_5 \wedge \text{r}_9 = \text{r}_5 \wedge \text{q}_9 = \text{q}_5)))$

Example Obligations

2.2 $\neg \mathrm{q\_is\_empty}_1 \Rightarrow \mathrm{q}_2 \neq \Lambda$

6.2 $(\neg \mathrm{q\_is\_empty}_1 \wedge \neg \mathrm{q\_is\_empty}_5) \Rightarrow \mathrm{q}_6 \neq \Lambda$

11.2 $|\mathrm{r}_{11}| \leq |\mathrm{c}_{11}| \wedge |\mathrm{c}_{11}| \leq |\mathrm{r}_{11}| + 1 \wedge \mathrm{IS\_PERMUT}(\mathrm{c}_{11} * \mathrm{r}_{11}, \mathrm{q}_0)$

The Indexed Method

1. The branch condition forms the antecedent of each fact and obligation.

2. From each procedure call we have a fact and an obligation.

   (a) The fact comes from the postcondition.

   (b) The obligation comes from the precondition.

3. From each selection statement we have a fact—the conjunction of two clauses; each is an antecedent followed by a conjunction of equalities.

4. From each while loop we have two facts and two obligations.

   (a) Obligation: the invariant must hold when the loop is first encountered.

   (b) Fact: the loop test and the invariant hold at the beginning of the loop body.

   (c) Obligation: the invariant must hold at the end of the loop body.

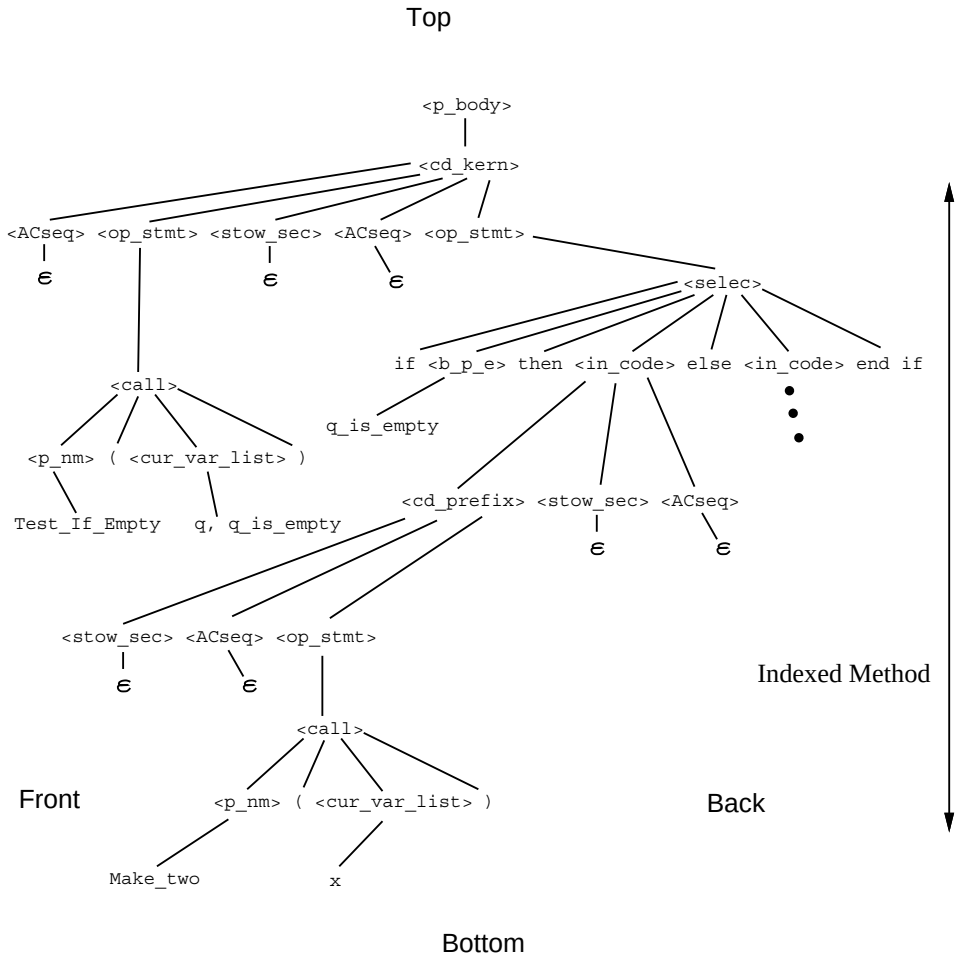
   (d) Fact: immediately following the loop, the invariant and the negation of the loop test hold.

5. We have shown the program correct if we prove each obligation using only the facts that precede it.

Anant Mittal, West Virginia U., has written a program that generates verification conditions according to the indexed method.

Action Directions

Differ Among Methods



Back Substitution Method

Symbolic Execution (Forward Accumulation Method)

Computer Program Verification: Improvements for Human Reasoning

Wayne D. Heym

November 16, 1995

The Thesis

1. The traditional formal method of reasoning about the behavior of programs is not natural.

2. There is a sound formal basis for a more natural reasoning method.

3. The soundness of this new formal basis is strong in the sense that the method is also logically complete (relative to the (in)completeness of the mathematics used).

1. The traditional formal method of reasoning about the behavior of programs is not natural.

2. There is a sound formal basis for (the partial-correctness portion of) a more natural reasoning method.

3. The soundness of this new formal basis is strong in the sense that the method is also logically complete (relative to the (in)completeness of the mathematical theories used in the program's behavioral specification and explanation).

Example

How We Want To Reason About Programs

Queue Operations

procedure Test_If_Empty (q: Queue
empty: Boolean)
ensures "$(\#q = q) \wedge (q = \Lambda \Leftrightarrow empty)$"
procedure Dequeue (q: Queue
x: Item)
requires "$q \neq \Lambda$"
ensures "$\#q = \langle x \rangle * q$"
procedure Enqueue (q: Queue
x: Item)
ensures "$q = \#q * \langle \#x \rangle \wedge$ **is_initial**(x)"

1. Implementation of merge sort using queues.

2. An earlier version of the queue template that supplies Test_If_Empty rather than Length as a primary operation.

3. Specified in terms of string theory.

4. The Dequeue operation has a nontrivial precondition: the queue must not be empty.

Body of Split

remember
assume $r = \Lambda \wedge c = \Lambda$
Test_If_Empty(q, q_is_empty)
loop
  maintaining $(q \neq \Lambda \Rightarrow |r| = |c|) \wedge |r| \leq |c| \wedge$
    $|c| \leq |r| + 1 \wedge$
    $\text{IS\_PERMUT}(c * q * r, \#c * \#q * \#r) \wedge \ \ldots$
  while not q_is_empty **do**
    Dequeue(q, x)
    Enqueue(c, x)
    Test_If_Empty(q, q_is_empty)
    if not q_is_empty **then**
      Dequeue(q, x)
      Enqueue(r, x)
    end if
    Test_If_Empty(q, q_is_empty)
end loop
confirm $|r| \leq |c| \wedge |c| \leq |r| + 1 \wedge$
    $\text{IS\_PERMUT}(c * r, \#q)$

1. The purpose of merge sort's split operation is to produce two queues of nearly equal length that contain the same items as were in the original queue.

2. Here we split the queue q into queues c (for catalyst) and r (comes after q in the alphabet).

3. We assume c and r start empty.

4. We want to make sure the lengths of c and r are roughly equal and that their concatenation is a permutation of the original q.

5. The loop invariant is constructed to imply our desired conclusion; it may need to include other conjuncts to help us along. For example, we need the extra information that the lengths of c and r are exactly equal when the queue q still contains items.

6. This implementation alternately gives one item of q to c and to r.

7. We would like to talk about the values of these variables at the different points in the program; so we mark the between-statement spaces with convenient indexes: integers.

```
– – (0)
  assume r = Λ ∧ c = Λ
  Test_If_Empty(q, q_is_empty)
– – (1)
  loop
    maintaining (q ≠ Λ ⇒ |r| = |c|) ∧ |r| ≤ |c| ∧  . . .
    while not q_is_empty do
– – (2)
      Dequeue(q, x)
– – (3)
      Enqueue(c, x)
– – (4)
      Test_If_Empty(q, q_is_empty)
– – (5)
      if not q_is_empty then
– – (6)
        Dequeue(q, x)
– – (7)
        Enqueue(r, x)
– – (8)
      end if
```

– – (9)

Test_If_Empty(q, q_is_empty)

– – (10)

end loop

– – (11)

confirm $|\mathrm{r}| \leq |\mathrm{c}| \wedge |\mathrm{c}| \leq |\mathrm{r}| + 1 \wedge \ \ldots$

Note that positions 2 and 10 are, respectively, at the beginning and the end of the loop body.

Example Reasoning Goals

- $(q_1 \neq \Lambda \wedge q_5 \neq \Lambda) \Rightarrow$
  $(c_{10} = c_2 * \langle \text{first}(q_2) \rangle$
  $\wedge\, r_{10} = r_2 * \langle \text{second}(q_2) \rangle)$

- $(q_1 \neq \Lambda \wedge q_5 = \Lambda) \Rightarrow$
  $(c_{10} = c_2 * \langle \text{first}(q_2) \rangle$
  $\wedge\, r_{10} = r_2)$

1. Along the way to showing that the loop body maintains the invariant, we want to establish some intermediate goals.

2. In case the loop body is executed (q_1 is not empty) and the then-part of the if statement is executed (q_5 is not empty), we want to show that c_{10} is the extension of c_2 by the first item of q_2 and that r_{10} is the extension of r_2 by the second item of q_2.

3. In case the loop body is executed and the then-part of the if statement is not executed, we want to show that c_{10} is the extension of c_2 by the first item of q_2 and that $r_{10} = r_2$.

4. We get to use some known facts to establish the truth of these and other goals.

Example Facts

1.1 $(\text{q\_is\_empty}_1 \Leftrightarrow \text{q}_1 = \Lambda) \wedge \ \ldots$

2.1 $\neg\text{q\_is\_empty}_1 \Rightarrow$
$(\neg\text{q\_is\_empty}_2 \wedge \ \ldots$
$\wedge (\text{q\_is\_empty}_2 \Leftrightarrow \text{q}_2 = \Lambda))$

3.1 $\neg\text{q\_is\_empty}_1 \Rightarrow$
$(\langle \text{x}_3 \rangle * \text{q}_3 = \text{q}_2 \wedge \text{r}_3 = \text{r}_2 \wedge \text{c}_3 = \text{c}_2)$

4.1 $\neg\text{q\_is\_empty}_1 \Rightarrow$
$(\text{c}_4 = \text{c}_3 * \langle \text{x}_3 \rangle \wedge \text{r}_4 = \text{r}_3 \wedge \text{q}_4 = \text{q}_3)$

9.1 $\neg\text{q\_is\_empty}_1 \Rightarrow (\ \ldots \wedge$
$(\text{q\_is\_empty}_5 \Rightarrow$
$(\text{x}_9 = \text{x}_5 \wedge \text{c}_9 = \text{c}_5 \wedge \text{r}_9 = \text{r}_5 \wedge \text{q}_9 = \text{q}_5)))$

1. We know 1.1 from the postcondition of Test_If_Empty.

2. We know 3.1 from the branch condition associated with the loop, the postcondition of Dequeue, and that Dequeue is specified as not changing any variable other than its parameters. Our reasoning about correctness is relative to the correctness of Queue Template's implementation.

3. We know 4.1 from the postcondition of Enqueue.

4. Fact 9.1 expresses that, when the test of the if statement evaluates as false, all variables have values at the end of the if statement the same as at its beginning.

5. Note that our facts deal with the variable q_is_empty but our goals deal with whether the queue q equals Lambda, or is empty. This makes us want to add another conjunct to our loop invariant.

6. Fact 2.1 comes from the loop invariant and the negation of the loop test.

Additional Conjunct for Invariant

$$\ldots \wedge (\text{q\_is\_empty} \Leftrightarrow \text{q} = \Lambda)$$

Example Obligations

2.2 $\neg \mathrm{q\_is\_empty}_1 \Rightarrow \mathrm{q}_2 \neq \Lambda$

6.2 $(\neg \mathrm{q\_is\_empty}_1 \wedge \neg \mathrm{q\_is\_empty}_5) \Rightarrow \mathrm{q}_6 \neq \Lambda$

11.2 $|\mathrm{r}_{11}| \leq |\mathrm{c}_{11}| \wedge |\mathrm{c}_{11}| \leq |\mathrm{r}_{11}| + 1 \wedge \mathrm{IS\_PERMUT}(\mathrm{c}_{11} * \mathrm{r}_{11}, \mathrm{q}_0)$

1. Aside from our intermediate goals, we incur specific obligations in showing this program correct. We have to show that the precondition of each procedure call is met.

2. Obligations 2.2 and 6.2 reflect operation Enqueue's precondition.

3. Obligation 11.2 reflects the final confirm statement of this program. In other words, it reflects the postcondition of the split operation for merge sort.

4. Each loop invariant demands two obligations and supplies two facts.

The Indexed Method

1. The branch condition forms the antecedent of each fact and obligation.

2. From each procedure call we have a fact and an obligation.

   (a) The fact comes from the postcondition.

   (b) The obligation comes from the precondition.

3. From each selection statement we have a fact—the conjunction of two clauses; each is an antecedent followed by a conjunction of equalities.

4. From each while loop we have two facts and two obligations.

   (a) Obligation: the invariant must hold when the loop is first encountered.

   (b) Fact: the loop test and the invariant hold at the beginning of the loop body.

   (c) Obligation: the invariant must hold at the end of the loop body.

   (d) Fact: immediately following the loop, the invariant and the negation of the loop test hold.

5. We have shown the program correct if we prove each obligation using only the facts that precede it.

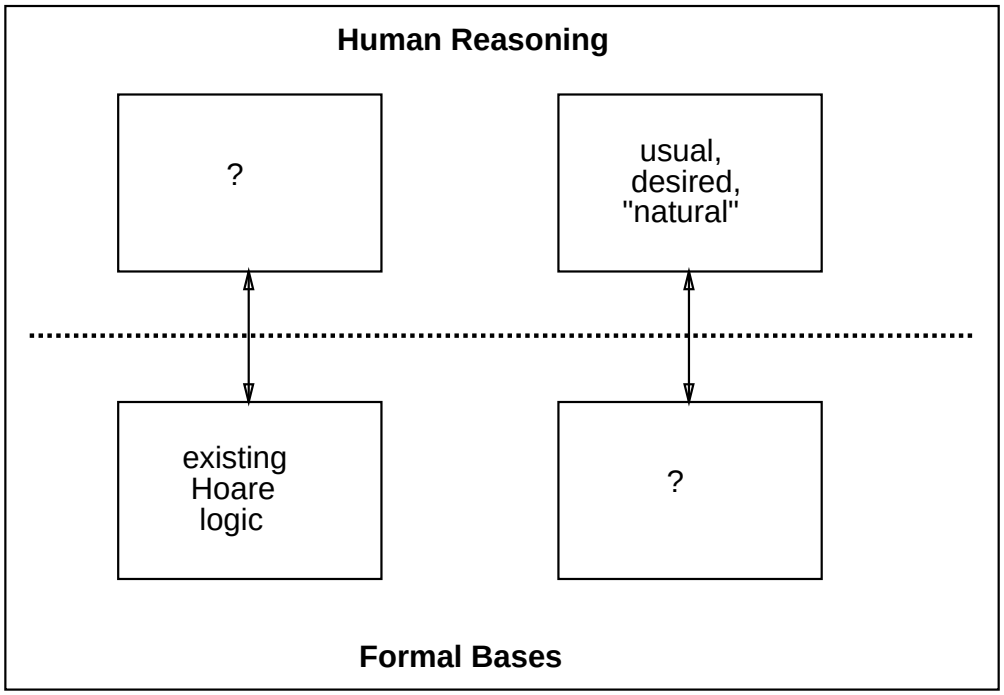
Anant Mittal, West Virginia U., has written a program that generates verification conditions according to the indexed method.

Existence of a verification condition generator means at least two things:

1. The problem of people forgetting to prove every obligation is alleviated.

2. It is proof of the principle of our mathematical analysis that the generation of verification conditions according to the indexed method is tractable.

Human Reasoning and Formal Bases



1. But is the indexed method correct?

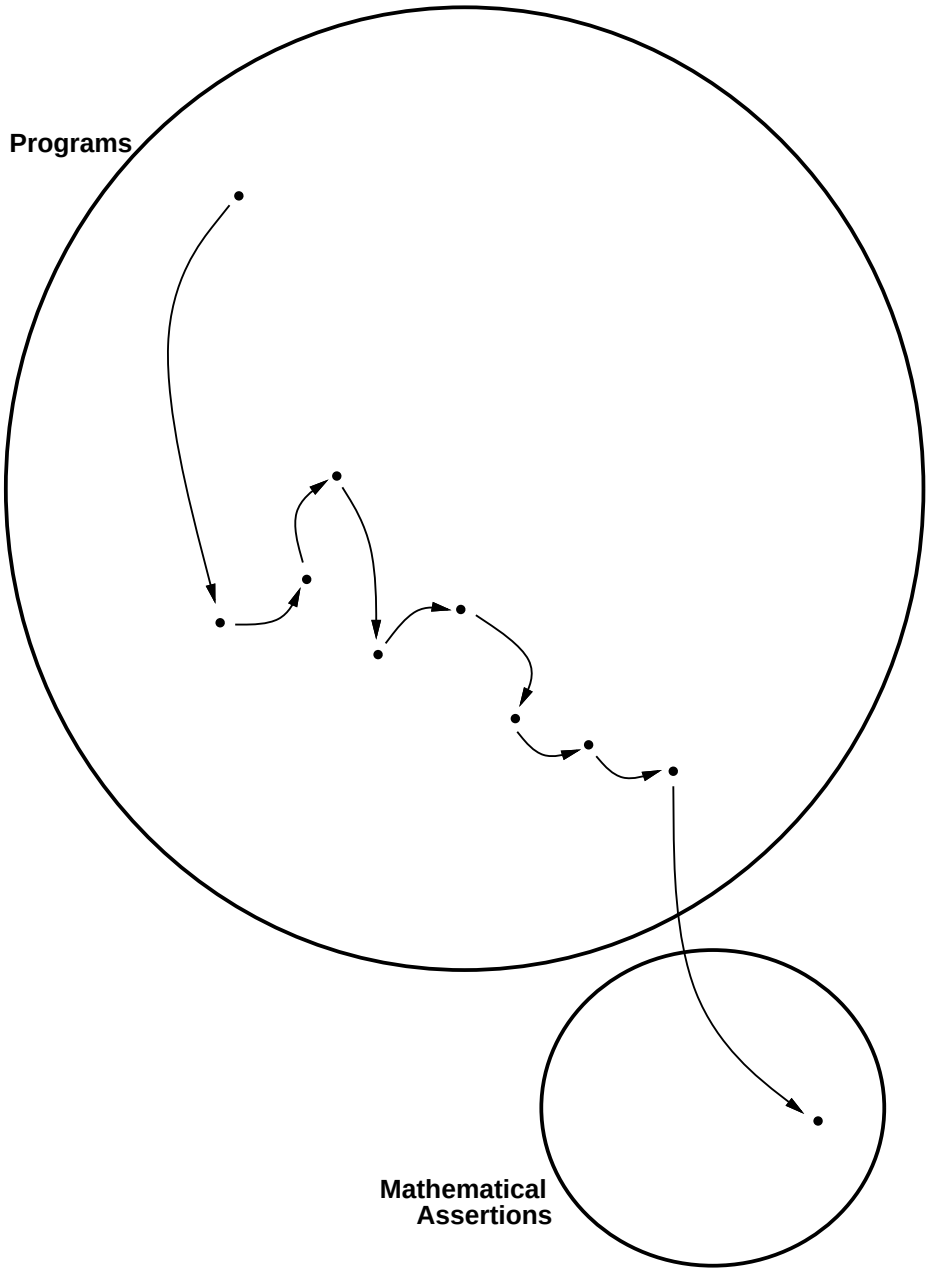
2. It is not clear what the human counterpart is for Hoare logic, the existing formal basis for reasoning about programs.

3. Usual human reasoning methods about programs lack a formal basis.

4. This dissertation supplies a formal basis for the indexed method, assuring us that it is correct.

Transforming Programs

To Mathematical Assertions



1. Formal bases for reasoning about programs are founded on the idea of transforming programs to mathematical assertions.

2. If the transformation (according to the rules) results in a true mathematical assertion, the program at the starting point is correct.

3. Alternatively, if we follow the arrows in the opposite direction, starting with a true mathematical assertion, any program we end up with is correct.

4. These are the good properties of a *sound* formal system.

5. If every correct program can be transformed to a true mathematical statement, the formal system is *relatively complete*.

Steps in Establishing the Thesis

1. Formalize the indexed method.
2. Show it to be sound.
3. Show it to be relatively complete.

Semantics

Denotational Semantics

- Current-states = (Current-variable-names $\rightarrow$ Values)
- Assert-statuses = $\{\mathrm{VT}, \mathrm{CF}, \mathrm{NL}\}$ (Navlakha, CWRU, 1978)
  - Vacuously True
  - Categorically False
  - Neutral

The program's state is augmented to an *environment*.

VT and CF are stuck states.

Denotational semantics is functional.

1. To formalize program correctness, we need a well-defined semantics for imperative programs that is amenable to mathematical argument; denotational semantics fits the bill.

2. A program state is a function from variable names to values; because we also have indexed variables, we modify these terms with the word "current."

3. I learned of assert status from Bill Ogden; the earliest reference I'm aware of is Jainendra Navlakha's 1978 dissertation out of Case Western Reserve University. I believe he was George Ernst's student.

4. If the initial state violates the program's precondition, the assert status becomes vacuously true. If the final state violates the program's postcondition, the assert status becomes categorically false. Navlakha generalized this notion to include internal assumptions and requirements.

5. An assert status that is neither vacuously true nor categorically false is neutral.

6. The program's environment includes at least the current state and the assert status.

7. Once the assert status is either VT or CF it stays there.

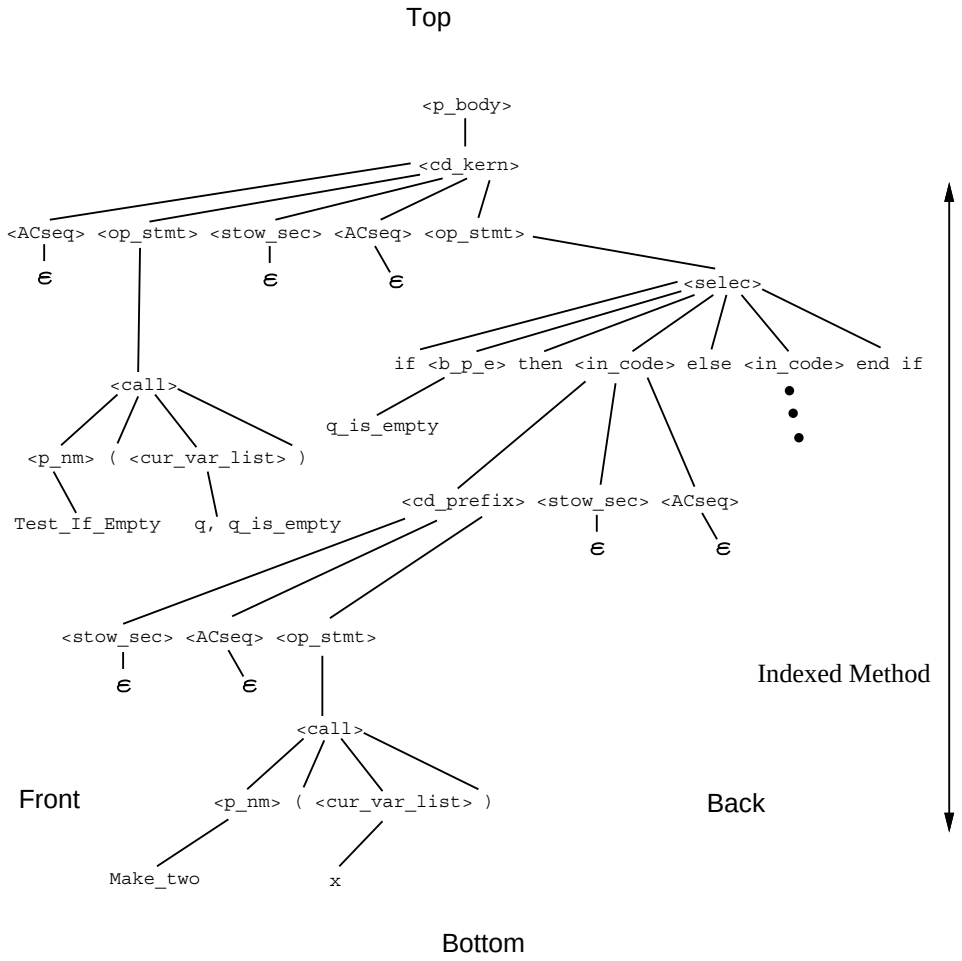
8. It will become important later to note that denotational semantics is a *function* from environments to environments.

Program Correctness: Validity

- A program is *valid* if and only if its execution in each neutral environment results in an environment that is not categorically false.
- A mathematical statement is *valid* if and only if it evaluates as true in every model of its theory.
- A proof system for assertive programs is *sound* if and only if every program that (using only the rules of the proof system) can be transformed to a valid mathematical statement is, itself, valid.

Action Directions

Differ Among Methods



Back Substitution Method

Symbolic Execution (Forward Accumulation Method)

1. The action directions of the various methods of proving program correctness differ one from the other.

2. The traditional methods can stay within the original programming language because they make the syntax tree narrower.

3. In fact, due to selection and iteration, the traditional methods produce multiple trees with their transformations.

4. The indexed method keeps the tree unified; it flattens the tree from top to bottom.

5. It does this by replacing operational statements (that may be interior to the tree) with assertions. This requires the transitional use of new programming language statements.

New PL Statements

- **stow**(i)

$$\text{Index-states} = (\text{Integers} \rightarrow \text{Current-states})$$

stow(i) changes the image of i under the index state to the value of the current state; i.e., it stores the current state in the ith component of the index state. It gives meaning to "$--(i)$".

- **whenever** Q **do** SS **end whenever**
  It behaves the same as the **if-then** statement; it gives meaning to "branch condition."

1. “Stow” means “store” or “save away”. I wanted to use a term different than the assembly language load and store mnemonics.

2. I chose the name “whenever” so it would be different from “if”.

New PL Statements

(continued)

- **alter all**

$$\text{Setups} = \text{Current-states}^* \text{ (Kleene star)}$$

alter all removes the first state from the setup and changes the current state to this value. Replacing executable statements with **assume**, **confirm**, and **alter all** statements enables the transformation from programs to assertions.

1. For example, a procedure call immediately precedes a **stow**(j) statement. It is replaced by a **confirm** statement and an **alter all** statement immediately preceding the **stow**(j) statement, and an **assume** statement immediately following the **stow**(j) statement.

2. The **confirm** statement changes a neutral environment to categorically false if the environment violates the procedure's precondition.

3. The **assume** statement changes a neutral environment to vacuously true if the current state resulting from the **alter all** statement violates the procedure's postcondition.

4. Hence, the **assume** statement *filters* out those environments resulting from setups that will not be informative.

Notation for Environment Named "env"

AE(env)	=	a	(Assert-status)
CSE(env)	=	cs	(Current-state)
ISE(env)	=	ns	(Index-state)
SPE(env)	=	se	(Setup)
OSE(env)	=	os	(Old-state)
DME(env)	=	d	(Declaration-meaning)

1. We have covered the first four components of an environment.

2. The old state is a function from identifiers preceded by one or more old signs to values. It handles remembering the current state from the beginning of a procedure or a loop.

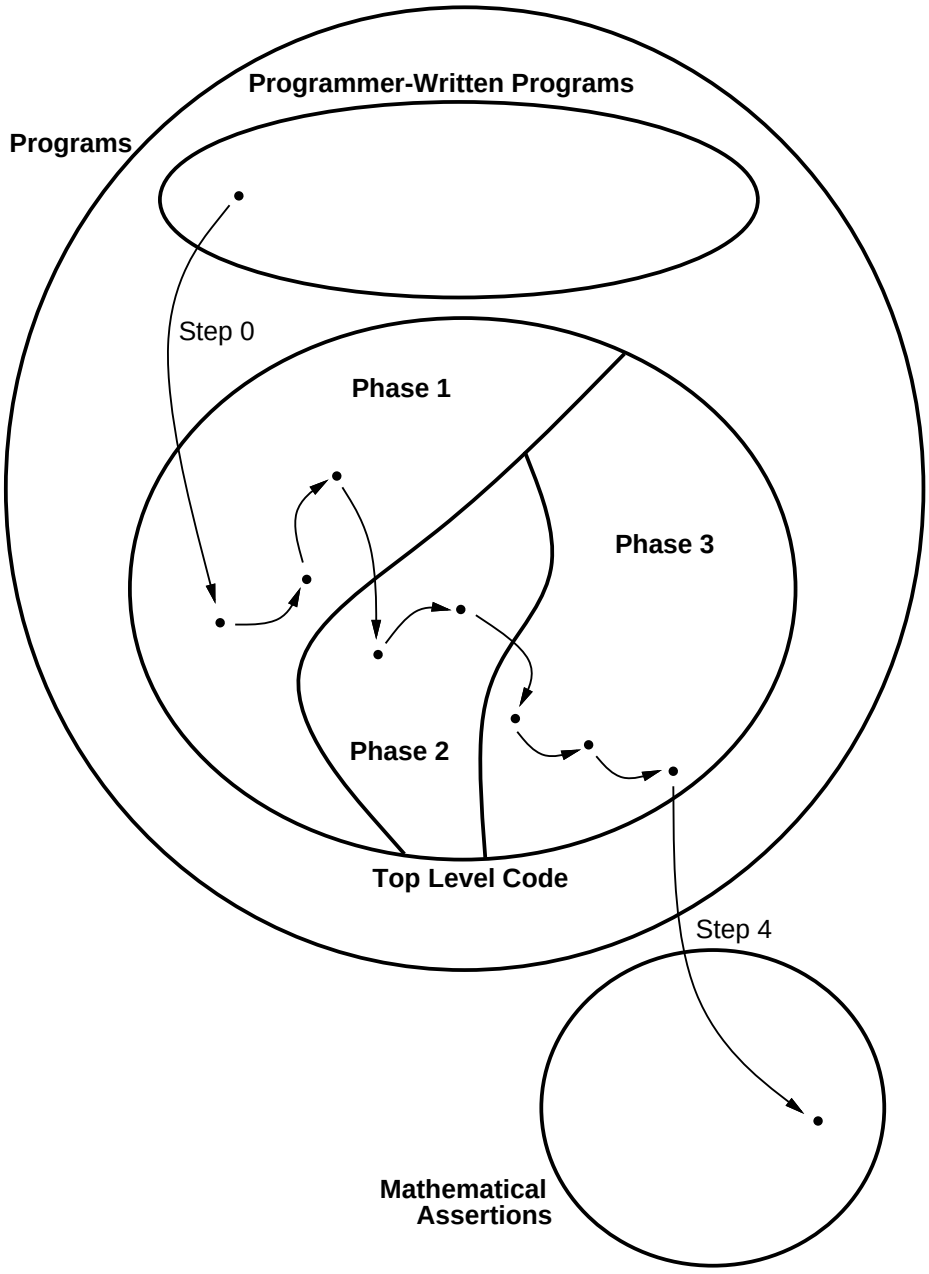
3. The declaration meaning is a function mapping identifiers into a rich range including type meanings and procedure meanings.

4. Environments that are missing meanings or that contain the wrong meanings are filtered out to vacuously true by program declarations.

5. On the left are the names of the projection functions; the typical exemplars are on the right.

Transforming Programs

To Mathematical Assertions in Phases



1. The rules of the indexed method constrain the transformation of programs in the math direction to occur in phases.

2. Step 0 is accomplished by the bridge rule.

3. Each step in Phase 1 removes a statement from the **whenever** statement containing it, or removes an empty **whenever** statement.

4. In Phase 2, there are no **whenever** statements. Here, pairs of **alter all stow** statements are removed.

5. Phase 3 programs are sequences of **assume** and **confirm** statements. They are consolidated to one **confirm** statement in this phase.

6. Step 4 produces the mathematical assertion contained in that one **confirm** statement.

To Prove Soundness

- We prove that each proof rule preserves *validity* when applied in the program direction.
- We *do not* prove that each proof rule preserves semantics.
- Using the contrapositive, we prove that each proof rule preserves *invalidity* when applied in the *math* direction.
- A program is *invalid* if and only if its execution in *some* neutral environment results in an environment that *is* categorically false.

For the third bullet item:

1. Preserving validity in the program direction means the following: if the more math-like entity is valid, then the resulting more program-like entity is also valid.

2. The contrapositive is: if the more program-like entity is invalid, then the more math-like entity is also invalid.

3. This latter is what we mean by preserving invalidity in the math direction.

Rules Maintain Restricted Syntax

1. Programmer-written programs do not contain **assume**, **stow**, **whenever**, and **alter all** statements. Their **confirm** statements refer to current variables only.
2. In the math direction, transformation begins with Joan Krone's rules (1988), including her procedure declaration rule.
3. The bridge rule produces *top level code*, a cycle of **alter all**, **stow**, **assume-confirm**, and **whenever** statements.
4. **whenever** statements contain no **alter all** statements, and *all* operational statements.
5. **assume** and **confirm** statements at the top level refer to indexed variables only.
6. Indexes are everywhere increasing.
7. References to index i occur only after **stow**(i).

1. Programmer-written program don't contain **assume** statements because all assumptions need to be either written in the specifications or be built in to the language.

Rule for alter all

$$\mathcal{P} = C\backslash \; prec\_top\_lev\_code \quad (1)$$
$$\textbf{alter all}$$
$$\textbf{stow}(i)$$
$$fol\_top\_lev\_code$$

$$\mathcal{M} = C\backslash \; prec\_top\_lev\_code \quad (2)$$
$$fol\_top\_lev\_code$$

Additional Syntactic Restriction:

$\mathcal{P}$ is a phase 2 program—one that contains no **whenever** statements.

1. The additional syntactic restriction is crucial.

2. The rule removes an **alter all stow** pair in the math direction.

3. We want to prove invalidity preservation in the math direction.

4. We suppose we have an environment that is a witness to $\mathcal{P}$'s invalidity. This environment is neutral and execution (or interpretation) of $\mathcal{P}$ in this environment produces a categorically false environment.

5. Take the case where the execution of the preceding top level code in this environment results in a neutral environment.

Rule for alter all: Soundness

$$\mathrm{env}_1^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_1, \mathrm{ns}, \mathrm{se}, \mathrm{os}, \mathrm{d}] \tag{3}$$

$$\mathrm{env}_I^{\mathcal{P}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}_I, \mathrm{se}_I \circ \mathrm{se}, \mathrm{os}, \mathrm{d}] \tag{4}$$

$$\mathrm{env}_I^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_I, \mathrm{ns}'_I, \mathrm{se}_I \circ \mathrm{tail}(\mathrm{se}), \mathrm{os}, \mathrm{d}], \text{ where} \tag{5}$$

$$\mathrm{ns}'_I(h) = \begin{cases} \mathrm{ns}_I(h) & \text{if } h \neq i \\ \mathrm{first}(\mathrm{se}) & \text{if } h = i \end{cases} \tag{6}$$

$$\mathrm{env}_s^{\mathcal{P}} = [\mathrm{NL}, \mathrm{first}(\mathrm{se}), \mathrm{ns}', \mathrm{tail}(\mathrm{se}), \mathrm{os}, \mathrm{d}] \tag{7}$$

$$\mathrm{env}_1^{\mathcal{M}} = [\mathrm{NL}, \mathrm{cs}_1, \mathrm{ns}', \mathrm{tail}(\mathrm{se}), \mathrm{os}, \mathrm{d}] \tag{8}$$

At the end of the explanation:

1. The two environments differ only in the current state.
2. The only statements that could be affected by the current state that could occur in following top level code are **stow** statements.
3. If there are no **stow** statements in following top level code, the final two environments differ only in their current states. Therefore, their assert statuses are equal: both are categorically false.
4. Each **stow** statement is preceded by an **alter all** statement. Therefore, if there are any **stow** statements in following top level code, the final two environments are entirely equal. Therefore, their assert statuses are equal: both are categorically false.

Soundness

Theorem 1 *The indexed method is sound.*

Definition

A proof system for well-prepared assertive programs is *relatively complete* if and only if for every well-prepared valid assertive program Prog there exists a related valid program Prog$'$, which differs from Prog at most in the internal assertions (loop invariants and procedure specifications), such that Prog$'$ can be transformed (using only the rules of the proof system) to a valid mathematical statement.

Functional and Relational Semantics

The RSRG intends a *relational* semantics for procedure calls. Because a procedure's functional meaning is any arbitrary function that meets the relational specification, I had thought that functional semantics simulated relational semantics (at least at the same level of abstraction).

Discovery by Proof

However, while attempting to prove relative completeness for the procedure call rule, I learned that functional semantics do not exactly simulate relational semantics even at the same level of abstraction.

According to Functional Semantics

This assertive program is valid:

$$
\begin{array}{ll}
C\backslash & \textbf{assume true} \\
 & \text{Make\_1\_Or\_Minus\_1(x)} \\
 & \text{Make\_1\_Or\_Minus\_1(y)} \\
 & \textbf{confirm } \mathrm{x} + \mathrm{y} \neq 0 \\
 & \textbf{end whenever}
\end{array}
$$

where $C \supseteq \{\mathrm{x}, \mathrm{y} : \text{Integer}\} \cup$

$$
\left\{ \begin{array}{l}
\textbf{procedure } \text{Make\_1\_Or\_Minus\_1(z: Integer)} \\
\quad \textbf{ensures } (\mathrm{z} = 1) \vee (\mathrm{z} = -1)
\end{array} \right\}
$$

Expressiveness

Cook (1978) defines the assertion language to be *expressive* if, for every precondition and statement sequence, there is a formula of the language that expresses the post relation corresponding to the precondition and statement sequence.

Relative Completeness

Theorem 2 *If the assertion language is expressive, then the indexed method is relatively complete over the set of programs that contain no calls to relationally specified external procedures.*

Contributions

- A formal basis for a reasoning method that supports as-needed, as well as systematic, strategies.
- New proof techniques: we are now able to prove validity preservation for top-to-bottom, tree-flattening processes.
  - Extended the use of assert status.
  - Indexed variables permit parallel processing of alternatives.
  - Setup with **alter all** permits unordered replacement of operational statements.
- Proofs are actually written down in detail.
- Stronger case for development of relational semantics.

1. Indexed variables permit us to keep the syntax tree unified during the transformation process.

2. The enhanced semantics using the richer environment and the additional programming language statements made it possible to write down the proofs in detail.

Future Work

- Develop relational semantics for assertive programs.
- Prove the indexed method sound and relatively complete with respect to relational semantics.
- Study human use of the indexed method empirically to validate its advantages.
- Investigate automated tool support; explore advantages/disadvantages.
- Formalize other programming constructs, e.g., function calls and **loop exit when**.
- Remove setup from environment; replace **alter all stow**(i) with **start from**(i).

1. Why validate the advantages of the indexed method if it is unsound?

Proof

Proof
Some people gonna call you up
Tell you something that you already know
Proof
Sane people go crazy on you
Say "No man, that was not
The deal we made
I got to go, I got to go"
Faith
Faith is an island in the setting sun
But proof, yes
Proof is the bottom line for everyone

I was pleased to discover that my favorite singer-songwriter has expressed the status of proof in song.

$$\mathcal{P} \stackrel{\text{def}}{=} C\backslash \quad prec\_top\_lev\_code \qquad (9)$$

alter all

stow(i)

$ACseq_0$

whenever Br_Cd **do**

loop

maintaining Inv$\lfloor$x, #x$\rfloor$

while b_p_e **do**

stow(j)

cd_kern

stow(k)

$ACseq$

end loop

stow(l)

cd_suffix

end whenever

$fol\_top\_lev\_code$