CHAPTER 2

Underlying Technologies

> "According to the grace of God which was given to me, as a wise master builder I have laid the foundation."
>
> — 1 Corinthians 3:10

Every building has a foundation. The foundation of this dissertation is the collection of technologies that underlie the work. This chapter will describe these technologies, and give a sense of how these elements fit together to support the goal of program verification.

To make this more concrete, we will take as an example a small programming language, similar to a subset of Pascal, with assignment statements, conditionals, and while loops. Associated with this language is a language of assertions, which describe conditions about states in the computer. For these languages, we will define their syntax and semantics, and give a Hoare logic as an axiomatic semantics for partial correctness. Using this logic, we will define a Verification Condition Generator for this programming language. Finally, we will discuss embedding this programming language and its VCG within Higher Order Logic.

This small programming language is not the language actually studied in

c	::=	**skip**
	|	**abort**
	|	$x := e$
	|	$c_1 \; ; \; c_2$
	|	**if** b **then** c_1 **else** c_2 **fi**
	|	**assert** a **while** b **do** c **od**

Table 2.1: Example programming language.

this dissertation, but in its simplicity serves as a clear illustration to discuss the fundamental technologies and ideas present in this chapter.

2.1 Syntax

Table 2.1 contains the syntax of a small programming language, defined using Backus–Naur Form as a context–free grammar. We denote the type of commands as `cmd`, with typical member c. We take as given a type of numeric expressions `exp` with typical member e, and a type of boolean expressions `bexp` with typical member b. We will further assume that these expressions contain all of the normal variables, constants, and operators.

These constructs are mostly standard. Informally, the **skip** command has no effect on the state. **abort** causes an immediate abnormal termination of the program. $x := e$ evaluates the numeric expression e and assigns the value to the variable x. $c_1 \; ; \; c_2$ executes command c_1 first, and if it terminates, then executes c_2. The conditional command **if** b **then** c_1 **else** c_2 **fi** first evaluates the boolean expression b; if it is true, then c_1 is executed, otherwise c_2 is executed. Finally, the iteration command **assert** a **while** b **do** c **od** evaluates b; if it is true,

then the body c is executed, followed by executing the whole iteration command again, until b evaluates to false. The '**assert** a' phrase of the iteration command does not affect its execution; this is here as an annotation to aid the verification condition generator. The significance of a is to denote an *invariant*, a condition that is true every time control passes through the head of the loop.

Annotations are written in an *assertion language* that is a partner to this programming language. The assertion language is used to expresses conditions that are true at particular moments in a program's execution. Usually these conditions are attached to specific points in the control structure, signifying that whenever control passes through that point, then the attached assertion evaluates to true. For this simple example, we will take the assertion language to be the first-order predicate logic with operators for the normal numeric and boolean operations. In particular, $a_1 => a_2 \mid a_3$ is a conditional expression, which first evaluates a_1, and then yields the value of a_2 or a_3 depending on whether a_1 was true or false, respectively. We also specifically include the universal and existential quantifiers, ranging over nonnegative integers. We denote the types of numeric expressions and boolean expressions in the assertion language as `vexp` and `aexp`, respectively, with typical members v and a. We will also use p and q occasionally as typical members of `aexp`.

We use the same operator symbols (like "+") in the programming and assertion languages, overloading the operators and relying on the reader to disambiguate them by context.

2.2 Semantics

The execution of programs depends on the state of the computer's memory. In this simple programming language, all variables have nonnegative integer values. Following the notation of HOL, we will denote the type of nonnegative integers by `num` and the type of truth values as `bool`. We take the type of variables to be `var`, without specifying them completely at this time. Then we can represent states as functions of type `state` = `var`→`num`, and we can refer to the value of a variable x in a state s as $s\ x$, using simple juxtaposition to indicate the application of a function to its argument.

Numeric and boolean expressions are evaluated by the curried functions E and B, respectively. Because these expressions may contain variables, their evaluation must refer to the current state.

$E\ e\ s = n$	Numeric expression e:`exp` evaluated in state s yields numeric value n:`num`.
$B\ b\ s = t$	Boolean expression b:`bexp` evaluated in state s yields truth value t:`bool`.

The notation $f[e/x]$ indicates the function f updated so that

$$(f[e/x])(y) = \begin{cases} e & \text{if } y = x, \text{ and} \\ f(y) & \text{if } y \neq x \end{cases}$$

The operational semantics of the programming language is expressed by the following relation:

$C\ c\ s_1\ s_2$	Command c executed in state s_1 yields resulting state s_2.

Skip:

$$\frac{}{C\ \textbf{skip}\ s\ s}$$

Abort:
(no rules)

Assignment:

$$\frac{}{C\ (x := e)\ s\ s[(E\ e\ s)/x]}$$

Sequence:

$$\frac{C\ c_1\ s_1\ s_2, \qquad C\ c_2\ s_2\ s_3}{C\ (c_1\ ;\ c_2)\ s_1\ s_3}$$

Conditional:

$$\frac{B\ b\ s_1\ = \mathrm{T}, \qquad C\ c_1\ s_1\ s_2}{C\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ s_1\ s_2}$$

$$\frac{B\ b\ s_1 = \mathrm{F}, \qquad C\ c_2\ s_1\ s_2}{C\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ s_1\ s_2}$$

Iteration:

$$\frac{B\ b\ s_1 = \mathrm{T}, \qquad C\ c\ s_1\ s_2 \quad C\ (\textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ s_2\ s_3}{C\ (\textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ s_1\ s_3}$$

$$\frac{B\ b\ s_1 = \mathrm{F}}{C\ (\textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ s_1\ s_1}$$

Table 2.2: Example programming language structural operational semantics.

Table 2.2 gives the structural operational semantics of the programming language, specified by rules inductively defining the relation C.

The semantics of the assertion language is given by recursive functions V and A defined on the structure of `vexp` and `aexp`, in a directly denotational fashion. Since the expressions may contain variables, their evaluation must refer to the current state.

$V\ v\ s = n$	Numeric expression v:`vexp` evaluated in state s yields numeric value n:`num`.
$A\ a\ s = t$	Boolean expression a:`aexp` evaluated in state s yields truth value t:`bool`.

This syntax and structural operational semantics is the foundational defini-

tion for this programming language and its meaning. It is complete, in that we know the details of any prospective computation, given the initial state and the program to be executed. However, it is not the easiest form with which to reason about the correctness of programs. For that, we need to turn to a more abstract representation of the semantics, such as Hoare-style program logics.

2.3 Partial and Total Correctness

When talking about the correctness of a program, exactly what is meant? In general, this describes the consistency of a program with its specification. There have developed two versions of the specific meaning of correctness, known as *partial correctness* and *total correctness*. *Partial correctness* signifies that every time you run the program, every answer that it gives you is consistent with what is specified. However, partial correctness admits the possibility of not giving you any answer at all, by permitting the possibility of the program not terminating. A program that does not terminate is still said to be partially correct. In contrast, *total correctness* signifies that every time you start the program, it will in fact terminate, *and* the answer it gives you will be consistent with what is specified.

The partial and total correctness of commands may be expressed by logical formulae called *Hoare triples*, each containing a precondition, a command, and a postcondition. The precondition and postcondition are boolean expressions in the assertion language. Traditionally, the precondition and postcondition are written with curly braces ($\{\ \}$) around them to signify partial correctness, and with square braces ($[\ ]$) to signify total correctness. For our example programming language and its assertion language, we define notations for partial and total correctness

$\{a\}$	$=$	**close** a $=$ $\forall s.\ A\ a\ s$
$\{p\}\ c\ \{q\}$	$=$	$\forall s_1\ s_2.\ A\ p\ s_1 \wedge C\ c\ s_1\ s_2 \Rightarrow A\ q\ s_2$
$[p]\ c\ [q]$	$=$	$(\forall s_1\ s_2.\ A\ p\ s_1 \wedge C\ c\ s_1\ s_2 \Rightarrow A\ q\ s_2)$
		$\wedge\ (\forall s_1.\ A\ p\ s_1 \Rightarrow (\exists s_2.\ C\ c\ s_1\ s_2))$

Table 2.3: Floyd/Hoare Partial and Total Correctness Semantics.

in Table 2.3.

As described in the table, we use $\{a\}$ to denote a boolean assertion expression which is true in all states. This is the same as having all of the free variables of a universally quantified, and so this is also known as the *universal closure* of a. **close** a denotes the same universal closure, but by means of a unary operator.

With these partial and total correctness notations, it now becomes possible to express an axiomatic semantics for a programming language, as a Hoare-style logic, which we will do in the next section.

In this dissertation, we will study a larger programming language that will include procedures with parameters. Verifying these procedures will introduce several new issues. It is an obvious but nevertheless significant feature that a procedure call has a semantics which depends on more than the syntactic components of the call itself—it must refer to the declaration of the procedure, which is external and part of the global context. This is unlike all of the constructs in the small example programming language given above.

The parameters to a procedure will include both value parameters, which are passed by value, and variable parameters, which are passed by name to simulate call-by-reference. The passing of these parameters, and their interaction with

global variables, has historically been a delicate issue in properly defining Hoare-style rules for the semantics of procedure call. The inclusion of parameters also raises the need to verify that no aliasing has occurred between the actual variables presented in each call and the global variables which may be accessed from the body of the procedure, as aliasing greatly complicates the semantics in an intractable fashion.

To verify total correctness, it is necessary to prove that every command terminates, including procedure calls. If the termination of all other commands is established, a procedure call will terminate unless it initiates an infinitely descending sequence of procedure calls, which continue issuing new calls deeper and deeper and never finishing them. To prove termination, we must prove this infinite recursive descent does not occur. This will constitute a substantial portion of this dissertation's work, as we describe a new method for proving the termination of procedure calls which we believe to be simpler, more general, and easier to use than previous proposals.

2.4 Hoare Logics

In [Hoa69], Hoare presented a way to represent the calculations of a program by a series of manipulations of logical formulae, which were symbolic representations of sets of states. The logical formulae, known as "axioms" and "rules of inference," gave a simple and beautiful way to express and relate the sets of possible program states at different points within a program. In fact, under certain conditions it was possible to completely replace a denotational or operational definition of the semantics of a language with this "axiomatic" semantics. Instead

Skip:

$$\frac{}{\{q\}\ \textbf{skip}\ \{q\}}$$

Abort:

$$\frac{}{\{\textbf{false}\}\ \textbf{abort}\ \{q\}}$$

Assignment:

$$\frac{}{\{q \lhd [e/x]\}\ x := e\ \{q\}}$$

Sequence:

$$\frac{\{p\}\ c_1\ \{r\},\quad \{r\}\ c_2\ \{q\}}{\{p\}\ c_1\ ;\ c_2\ \{r\}}$$

Conditional:

$$\frac{\{p \wedge b\}\ c_1\ \{q\} \qquad \{p \wedge \sim b\}\ c_2\ \{q\}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi}\ \{q\}}$$

Iteration:

$$\frac{\{a \wedge b\}\ c\ \{a\} \qquad \{a \wedge \sim b \Rightarrow q\}}{\{a\}\ \textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od}\ \{q\}}$$

Precondition Strengthening:

$$\frac{\{p \Rightarrow a\} \qquad \{a\}\ c\ \{q\}}{\{p\}\ c\ \{q\}}$$

Table 2.4: Example programming language axiomatic semantics.

of involving states, these "rules" now dealt with symbolic formulae representing sets of possible states. This had the benefit of more closely paralleling the reasoning needed to actually prove a program correct, without being as concerned with the details of actual operational semantics. To some, reasoning about states seemed "lower level" and more representation-dependent than reasoning about expressions denoting relationships among variables.

To illustrate these ideas, consider the Hoare logic in Table 2.4 for the simple programming language we have developed so far.

In the rule for Assignment, the precondition is $q \lhd [e/x]$. $\lhd$ denotes the operation of proper substitution; hence, this denotes the proper substitution of the expression e for the variable x throughout the assertion q. There is one small

problem with this, which is that the expressions e and q are really from two different, though related, languages. We will intentionally gloss over this issue now, simply using e as a member of both languages. This also applies to b where it appears in the Conditional and Iteration rules.

Given these rules, we may now compose them to prove theorems about structured commands. For example, from the Rule of Assignment, we have

$$\{x0 = 0 * y0 + x \wedge y0 = y\}\ r := x\ \{x0 = 0 * y0 + r \wedge y0 = y\}$$

and

$$\{x0 = 0 * y0 + r \wedge y0 = y\}\ q := 0\ \{x0 = q * y0 + r \wedge y0 = y\}.$$

From these and the Rule of Sequence, we have

$$\{x0 = 0 * y0 + x \wedge y0 = y\}\ r := x\ ;\ q := 0\ \{x0 = q * y0 + r \wedge y0 = y\}.$$

For completeness, a Hoare logic will usually contain additional rules not based on particular commands, such as precondition strengthening or postcondition weakening. The Precondition Strengthening Rule in Table 2.4 is an example.

2.5 Soundness and Completeness

An axiomatic semantics for a programming language has the benefit of better supporting proofs of program correctness, without involving the detailed and seemingly mechanical apparatus of operational semantics. However, with this benefit of abstraction comes a corresponding weakness. The very fact that the new Hoare rules are more distant from the operational details means a greater possibility that in fact they might not be logically consistent. This question

of consistency has two aspects, which are called *soundness* and *completeness*. *Soundness* is the quality that every rule in the axiomatic semantics is true for every possible computation described by the foundational operational semantics. A rule is sound if every computation that satisfies the antecedents of the rule also satisfies its consequent. *Completeness* is the quality of the axiomatic semantics of being expressive and powerful enough to be able to prove within the Hoare logic theorems that represent every computation allowed by the operational semantics. One could easily come up with a sound axiomatic semantics by having only a few trivial rules; but then one would never be able to derive useful results about interesting programs. Likewise, one could come up with powerful axiomatic semantics with which many theorems about programs could be proven; but if any one rule is not sound, the entire system is useless.

Of these two qualities, we have chosen for this dissertation to concentrate on soundness. By this choice, we do not intend to minimize the role or importance of completeness—it is simply a question of not being able to solve every problem at once. Nevertheless, we do feel that of the two qualities, soundness is in some sense the more vital one. A system that is sound but not complete may still be useful for proving many programs correct. A system that is complete but not sound will give you the ability to prove many seemingly powerful theorems about programs which are in fact not true with respect to the operational semantics.

Also, researchers have occasionally proposed rules for axiomatic semantics which were later found to be unsound. This problem has arisen, for example, in describing the passing of parameters in procedure calls. This history shows a need for some mechanism to more carefully establish the soundness of the rules

of an axiomatic semantics, thereby establishing the rules as trustworthy, since all further proof efforts in that language depend on them.

2.6 Verification Condition Generators

Given a Hoare logic for a particular programming language, it may be possible to partially automate the process of applying the rules of the logic to prove the correctness of a program. Generally this process is guided by the structure of the program, applying in each case the Hoare logic rule for the command which is the major structure of the phrase under consideration.

A Verification Condition Generator takes a suitably annotated program and its specification, and traces a proof of its correctness, according to the rules of the language's axiomatic semantics. Each command has its own appropriate rule which is applied when that command is the major structure of the current proof goal. This replaces the current goal by the antecedents of the Hoare rule. These antecedents then become the subgoals to be resolved by further applications of the rules of the logic.

At certain points, the rules require that additional conditions be met; for example, in the Iteration Rule in Table 2.4, there is the antecedent $a \wedge \sim b \Rightarrow q$. This is not a partial correctness formula, and so cannot be reduced further by rules of the Hoare logic. The VCG emits this as a verification condition to be proven by the user.

As an example, we present in Table 2.5 a Verification Condition Generator for the simple programming language discussed so far. It consists of two functions,

$vcg1$	$vcg1\ (\textbf{skip})\ q = q, [\,]$ $vcg1\ (\textbf{abort})\ q = \textbf{true}, [\,]$ $vcg1\ (x := e)\ q = q \lhd [e/x], [\,]$ $vcg1\ (c_1\,;\,c_2)\ q = \textbf{let}\ (r, h_2) = vcg1\ c_2\ q\ \textbf{in}$ $\textbf{let}\ (p, h_1) = vcg1\ c_1\ r\ \textbf{in}$ $p,\ (h_1\ \&\ h_2)$ $vcg1\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ q =$ $\textbf{let}\ (r_1, h_1) = vcg1\ c_1\ q\ \textbf{in}$ $\textbf{let}\ (r_2, h_2) = vcg1\ c_2\ q\ \textbf{in}$ $(b\ =>\ r_1\ \mid\ r_2),\ (h_1\ \&\ h_2)$ $vcg1\ (\textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ q =$ $\textbf{let}\ (p, h) = vcg1\ c\ a\ \textbf{in}$ $a,\ [a \wedge b \Rightarrow p\,;$ $a\ \wedge \sim b \Rightarrow q]\ \&\ h$
vcg	$vcg\ p\ c\ q = \textbf{let}\ (r, h) = vcg1\ c\ q\ \textbf{in}\ [p \Rightarrow r]\ \&\ h$

Table 2.5: Example Verification Condition Generator.

the main function *vcg* and a helper function *vcg*1. The square brackets [and] enclose a list, for which semicolons (';') separate list elements; the phrase [] denotes an empty list. Comma (',') creates a pair, and ampersand ('&') appends two lists together.

*vcg*1 has type `cmd`→`aexp`→(`aexp` × `(aexp)list`). This function takes a command and a postcondition, and returns a precondition and a list of verification conditions that must be proved in order to verify that command with respect to the precondition and postcondition. This function does most of the work of calculating verification conditions.

*vcg*1 is called by the main verification condition generator function, *vcg*, with type `aexp`→`cmd`→`aexp`→`(aexp)list`. *vcg* takes a precondition, a command, and a postcondition, and returns a list of the verification conditions for that command.

Given such a Verification Condition Generator, there are two interesting things we might ask about it. First, does the truth of the verification conditions it generates in fact imply the correctness of the program? If so, then we say the VCG is *sound*. Second, if the program is in fact correct, does the VCG generate verification conditions sufficient to prove the program correct from the axiomatic semantics? We call such a VCG *complete*. In this dissertation, we will only focus on the first question, that of soundness.

2.7 Higher Order Logic

Higher Order Logic (HOL) is a mechanical proof assistant that mechanizes higher order logic, and provides an environment for defining systems and proving state-

ments about them. It is secure in that only true theorems may be proven, and this security is ensured at each point that a theorem is constructed.

HOL has been applied in many areas. The first and still most prevalent use is in the area of hardware verification, where it has been used to verify the correctness of several microprocessors. In the area of software, HOL has been applied to Lamport's Temporal Logic of Actions (TLA), Chandy and Misra's UNITY language, Hoare's CSP, and Milner's CCS and π-calculus. HOL is one of the oldest and most mature mechanical proof assistants available, roughly comparable in maturity and degree of use with the Boyer-Moore Theorem Prover [BM88]. Many other proof assistants have been introduced more recently that in some ways surpass HOL, but HOL has one of the largest user communities and history of experience. We therefore considered it ideal for this work.

HOL differs from the Boyer-Moore Theorem Prover in that HOL does not attempt to automatically prove theorems, but rather provides an environment and supporting tools to the user to enable him to prove the theorems. Thus, HOL is better described as a mechanical proof assistant, recording the proof efforts and its products along the way, and maintaining the security of the system at each point, but remaining essentially passive and directed by the user. It is, however, powerfully programmable, and thus the user is free to construct programs which automate whatever theorem-proving strategy he desires.

2.7.1 Higher Order Logic as a Logic

Higher Order Logic is a version of predicate calculus which allows quantification over predicate and function symbols of any order. It is therefore an *ω-order*

logic, or *finite type theory,* according to Andrews [And86]. In such a type theory, all variables are given types, and quantification is over the values of a type. Type theory differs from set theory in that functions, not sets, are taken as the most elementary objects. Some researchers have commented that type theory seems to more closely and naturally parallel the computations of a program than set theory. A formulation of type theory was presented by Church in [Chu40]. Andrews presents a modern version in [And86] which he names $\mathcal{Q}_0$. The logic implemented in the Higher Order Logic system is very close to Andrews' $\mathcal{Q}_0$. This logic has the power of classical logic, with an intuitionistic style. The logic has the ability to be extended by several means, including the definition of new types and type constructors, the definition of new constants (including new functions and predicates), and even the assertion of new axioms.

The HOL logic is based on eight rules of inference and five axioms. These are the core of the logical system. Each rule is sound, so one can only derive true results from applying them to true theorems. As the HOL system is built up, each new inference rule consists of calls to previously defined inference rules, ultimately devolving to sequences of these eight primitive inference rules. Therefore the HOL proof system is fundamentally sound, in that only true results can be proven.

HOL provides the ability to assert new axioms; this is done at the user's discretion, and he then bears any responsibility for possible inconsistencies which may be introduced. Since such inconsistencies may be hard to predict intuitively, we have chosen in our use of the HOL system to restrict ourselves to never using the ability to assert new axioms. This style of using HOL is called a "definitional" or "conservative extension," because it is assured of never introducing any incon-

sistencies. In a conservative extension, the security of HOL is not compromised, and hence the basic soundness of HOL is maintained.

We will not describe in detail the theoretical foundation of the HOL logic, referring the interested reader to [GM93], because the purpose of this dissertation is not the study of HOL itself, but rather its application as a tool to support the verification of VCGs. Hence we will concentrate on describing the useful aspects of HOL that apply to our work.

2.7.2 Higher Order Logic as a Mechanical Proof Assistant

The HOL system provides the user a logic that can easily be extended, by the definition of new functions, relations, and types. These extensions are organized into units called *theories*. Each theory is similar to a traditional theory of logic, in that it contains definitions of new types and constants, and theorems which follow from the definitions. It differs from a traditional theory in that a traditional theory is considered to contain the infinite set of all possible theorems which could be proven from the definitions, whereas a theory in HOL contains only the subset which have been actually proven using the given rules of inference and other tools of the HOL system.

When the HOL system it started, it presents to the user an interactive programming environment using the programming language ML, or *Meta Language* of HOL. The user types expressions in ML, which are then executed by the system, performing any side effects and printing the value yielded. The language ML contains the data types `term` and `thm`, which represent terms and theorems in the HOL logic. These terms represent a second language, called the *Object Language*

(OL) of HOL, embedded within ML. ML functions are provided to construct and deconstruct terms of the OL language. Theorems, however, may not be so freely manipulated. Of central importance is the fact that theorems, objects of type `thm`, can only be constructed by means of the eight standard rules of inference. Each rule is represented as a ML function. Thus the security of HOL is maintained by implementing `thm` as an abstract data type.

Additional rules, called *derived rules of inference*, can be written as new ML functions. A derived rule of inference could involve thousands of individual calls to the eight standard rules of inference. Each rule typically takes a number of theorems as arguments and produces a theorem as a result. This methodology of producing new theorems by calling functions is called *forward proof*.

One of the strengths of HOL is that in addition to supporting forward proof, it also supports *backwards proof*, where one establishes a goal to be proved, and then breaks that goal into a number of subgoals, each of which is refined further, until every subgoal is resolved, at which point the original goal is established as a theorem. At each refinement step, the operation that is applied is called in HOL a *tactic*, which is a function of a particular type. The effect of applying a tactic is to replace a current goal with a set of subgoals which if proven are sufficient to prove the original goal. The effect of a tactic is essentially the inversion of an inference rule. Tactics may be composed by functions called *tacticals*, allowing a complex tactic to be built to prove a particular theorem.

Functions in ML are provided to create new types, make new definitions, prove new theorems, and store the results into theories on disk. These may then be used to support further extensions. In this incremental way a large system may

be constructed.

2.8 Embeddings

Previous researchers have constructed representations of programming languages within HOL, of which the work of Gordon [Gor89] was seminal. He introduced new constants in the HOL logic to represent each program construct, defining them as functions directly denoting the construct's semantic meaning. This is known as a "shallow" embedding of the programming language in the HOL logic, using the terminology described in [BGG+92]. This approach yielded tools which could be used to soundly verify individual programs. However, there were certain fundamental limitations to the expressiveness of this approach, and to the theorems which could be proven about all programs. This was recognized by Gordon himself [Gor89]:

> $\mathcal{P}[\mathcal{E}/\mathcal{V}]$ (substitution) is a meta notation and consequently the assignment axiom can only be stated as a meta theorem. This elementary point is nevertheless quite subtle. In order to prove the assignment axiom as a theorem within higher order logic it would be necessary to have types in the logic corresponding to formulae, variables and terms. One could then prove something like:
>
> $\vdash \forall P\ E\ V.\, \mathsf{Spec}\,(\mathsf{Truth}\,(\mathsf{Subst}\,(P, E, V)), \mathsf{Assign}\,(V, \mathsf{Value}\,E), \mathsf{Truth}\,P)$
>
> It is clear that working out the details of this would be a lot of work.

This dissertation explores the alternative approach described but not investigated by Gordon. It yields great expressiveness and control in stating and prov-

ing as theorems within HOL concepts which previously were only describable as meta-theorems outside HOL. For example, we have proven the assignment axiom described above:

$$\vdash \forall q\ x\ e.\ \{q \lhd [x := e]\}\ x := e\ \{q\}$$

where $q \lhd [x := e]$ is a substituted version of q, described later.

To achieve this expressiveness, it is necessary to create a deeper foundation than that used previously. Instead of using an extension of the HOL Object Language as the programming language, we create an entirely new set of datatypes within the Object Language to represent constructs of the programming language and the associated assertion language. This is known as a "deep" embedding, as opposed to the shallow embedding developed by Gordon. This allows a significant difference in the way that the semantics of the programming language is defined. Instead of defining a construct *as* its semantics meaning, we define the construct as simply a syntactic constructor of phrases in the programming language, and then separately define the semantics of each construct in a structural operational semantics. This separation means that we can now decompose and analyze syntactic program phrases at the HOL Object Language level, and thus reason within HOL about the semantics of purely syntactic manipulations, such as substitution or verification condition generation, since they exist *within* the HOL logic.

This has definite advantages because syntactic manipulations, when semantically correct, are simpler and easier to calculate. They encapsulate a level of detailed semantic reasoning that then only needs to be proven once, instead of having to be repeatedly proven for every occurrence of that manipulation. This

will be a recurring pattern in this dissertation, where repeatedly a syntactic manipulation is defined, and then its semantics is described, and proved correct within HOL.