DATA TYPES AND ALIASING

IN PROGRAM SPECIFICATION AND VERIFICATION

Joseph N. Wilson, Ph.D.
University of Virginia

Computer Science Report No. TR-86-13
May 23, 1986

Data Types and Aliasing
In Program Specification and Verification

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Joseph N. Wilson

May, 1985

Abstract

Programs in which aliasing occurs have been shown to be difficult to analyze. In particular the goal of formal verification of programs is hindered by the complexity which aliasing introduces. Most works on formal verification have ignored aliasing entirely. Similarly data types have largely been ignored in program verification work.

This work presents a syntactic and semantic definition of the programming language Hg, based on hierarchical graphs (h–graphs), and in which a large class of aliases can occur. We define the pertinent properties of h–graphs and present a *sound* and *relatively complete* verification system for the language. The method of proof we give is capable of expressing all true properties of programs involving even the most complex cases of aliasing. It does, however, also prove to be cumbersome. We introduce a formal system of type specification for Hg defining the notion of type correctness and the special case of static type correctness. We show how to incorporate this notion of data type into a proof system so that a proof can be simplified significantly when the type structure of the program in question exhibits certain characteristics. In particular, we can simplify the proofs of programs where type structure prohibits some aliases to occur.

Acknowledgements

There are many whom I should thank for helping me in completing this work. My committee has my gratitude for their perseverence in assessing the merit of this work. Of course, those who made this work possible by passing before me in the abstract world of program verification must be credited. In particular I must acknowledge Jaco de Bakker and Derek Oppen because, though I have never met them, I have felt their presence in many recesses of the world of program verification. I also thank the students who have taken T.W. Pratt's seminars and helped me via discussions of the many subtleties involved in the use of h–graphs. Many discussions with Alex Colvin, related and unrelated to my dissertation topic, helped me crystallize abstract concepts which made this work possible. And of course Dave Stotts has suffered through the same trials and tribulations as I: always shooting at a moving research target; attempting to get troff, pic, eqn, refer, and other beastly creations to work for man and not *vice–versa*; trying to finish a degree yet remain a responsive human being on this most wonderful of planets. But two above all others are to be thanked: the two Terrys in my life — Terry Pratt, the advisor and sculptor who has shaped more than just my research and shown me that the roles of metaphysician and scholar are not necessarily incompatible; and Terry Keenan, who can breathe a sigh of relief, knowing that although the journey is not ended, we can share a moment together knowing that perhaps the roughest part of the road has been traveled.

This research supported in part by National Science Foundation grant #MCS-7800763 and Grant N60921-81-K from the U. S. Naval Surface Weapons Center, Dahlgren, Virginia.

Table of Contents

List of Figures

List of Definitions

List of Theorems

List of Symbols

Symbol	*page*	*description*
$\mathbf{N}$	12	the natural numbers
$\mathbf{N}^+$	12	the whole numbers
$X^{\mathbf{N}}$	12	tuples over the set X
X^{ω}	13	sequences over the set X
$\bar{x}$	13	a sequence
$\bar{x}\downarrow i$	13	ith element of a tuple or sequence
$\|\bar{x}\|$	13	length of the tuple $\bar{x}$
$\bar{x}^\frown\bar{y}$	13	concatenation of $\bar{x}$ and $\bar{y}$
$A\hat{}\alpha$	13	singleton union
Φ	14	set of all nodes
Ξ	14	set of characters
Δ	14	the atomset
a	14	a member of Δ
$\perp$	14	the undefined value
Ω	15	the graph set
Gsel	16	the domain of graph selectors
gsel	16	a member of *Gsel*
$[\![gsel]\!]$	16	function denoted by a graph selector
Γ	17	the set of all H–graphs
nodeset(h)	17	nodeset of an h–graph
V^+	19	extended value function
Sel	19	the h–graph selectors
s	19	a member of *Sel*

$[\![s]\!]$	19	function denoted by a selector
Pred	27	the predicate symbols
Const	27	the constant symbols
Func	27	the function symbols
$\mathcal{T}$	27	the set of truthvalues
T	27	truthvalue *true*
F	27	truthvalue *false*
U	27	truthvalue *undefined*
$\mathcal{U}$	28	underlying logical structure
$\models_{\mathcal{U}} F$	28	validity of formula F in $\mathcal{U}$
Stat	31	the state set
σ	31	a member of *Stat*
$f\{v:n\}$	31	value substitution
$f\{\bar{v}:\bar{n}\}$	32	vector value substitution
$\sigma\{v:s\}$	32	variant of state
$\sigma\{\bar{v}:\bar{s}\}$	32	vector variant of state
E_n^m	32	node replacement
$E_{\bar{m}}^{n}$	32	vector node replacement
Expr	33	the domain of expressions
e	33	an element of *Expr*
R	33	the expression value function
Bool	33	the boolean expressions
b	33	an element of *Bool*
B	34	the boolean value function
Pname	34	procedure names
p	34	typical element of Proc
Stmt	34	the statements

k	34	member of *Stmt*
Code	34	statement sequences
K	34	member of *Code*
Pdef	35	procedure definitions
Pmap	35	procedure definition maps
π	35	typical element of *Pmap*
Prog	35	Hg programs
P	35	member of Prog
pnames	35	procedure names of program P
Last	39	last element of a state sequence
$Comp_P$	39	computation sequence function for P
Term	49	terms of assertions
d	49	member of *Term*
Assn	49	assertions
P	49	member of *Assn*
Q	49	member of *Assn*
Tvalue	50	term value function
$[\![P]\!]$	50	function denoted by a assertion
Form	50	correctness formulae
F	50	member of *Form*
$[\![F]\!]$	50	function denoted by a correctness formula
eq	51	node equality function
$\models F$	51	validity of correctness formula
$wp(K,Q)$	51	weakest precondition
$sp(K,P)$	52	strongest postcondition
P_s^e	54	Simple textual substitution
∇_s^r	58	substitution rule

$\overline{\nabla}$	61	substitution rule
Δ_s^e	68	substitution rule
$\Delta_{\bar{s}}^{\bar{e}}$	68	extension of Δ_s^e to vectors
$\frac{F_1, \cdots F_n}{F}$	83	inference rule
def	84	definite value function
$\vdash_{Ax,Pr} F$	87	F is provable
Ax	89	The Axiom set
Pr	89	The proof rules
$\nabla_{\bar{s}}^{\bar{r}}$	99	extension of ∇_s^r to vectors
Ax	105	The Axiom set extended
Pr	105	The proof rules extended
$\mathcal{G}$	111	h–graph grammar
τ	111	node type function
$\underset{\mathcal{G}}{\Rightarrow}$	112	direct derivation in a grammar
$\overset{*}{\Rightarrow}$	113	derivation in a grammar
$\mathcal{L}_{\mathcal{G}}$	113	language defined by a grammar
$\theta_{\mathcal{G}}$	115	grammar selector typeset function
$Pdef$	117	procedure definitions extended
$\mathcal{G}_c$	118	co–grammar of a grammar
$Prog$	118	h–graph programs
α	119	arity of functions
ϕ	119	types of function calls
θ^*	120	potential expression type
τ^*	120	actual type of an expression in a given state
$\sigma\{v,t:s\}$	120	typed variant of state
$Comp_P$	122	state sequence function for typed programs

$trim(h)$	123	h–graph h with inaccessible parts removed
$\models_{\mathcal{G}} F$	128	validity of correctness formula with respect to $\mathcal{G}$
$\vdash_{Ax,Pr,\mathcal{G}} F$	128	F is provable w.r.t. $\mathcal{G}$

Chapter 1

Aliasing and Program Verification

Aliasing is a central problem in attempts to formally verify the correctness of programs. Aliasing, informally defined, is what occurs when a single data object can be referenced by more than one name at a particular point during program execution. For example, after execution of the following Pascal program segment:

```
type list = record
        a, b : integer;
        end;
var p, q : ↑list;
begin
        new( p);
        q := p;
```

both of the names p↑.a and q↑.a refer to the same integer data object. Thus the names p↑.a and q↑.a are *aliases* from the time of execution of the statement q := p, until either p or q is assigned a different value.

Now suppose that at a later point in program execution, we encounter the assignment

```
p↑.a := 4;
```

If q↑.a is still an alias of p↑.a we know that the value of q↑.a after the assignment is 4. But if q↑.a is not an alias of p↑.a then the assignment does not affect the value of q↑.a. This statement, which seems to be quite simple, is actually difficult to understand in the presence of aliasing.

The goal of this study is to develop a deeper understanding of the role of aliasing in program verification. Several previous studies have considered the problem [1,2], but each

was hampered by an inadequate model of the data objects and aliasing structures possible in real programming languages. This study is based on a different model of data, the hierarchical graph, which allows the full complexity of data objects and referencing patterns in a language such as Pascal to be considered. Thus we are able to formally model arrays, records, pointers, linked lists (including circular lists), stacks, trees, and other common data objects, along with a full range of manipulations of these objects, including assignments of pointers and data objects containing pointers.

Hierarchical Graphs (h–graphs) have been used as a basis for programming language definition and program representation by Pratt since 1969 [3, 4, 5, 6]. In the past several years, the emphasis in h–graph research has changed from that of semantic specification of programming languages to the modeling of particular programs with hierarchical graphs. The programming language Hg provides a direct representation of programs based on hierarchical graph data types.

For the reader not familiar with h–graphs, we present an introduction to h–graphs. We also present, for the first time, a semantic specification of the programming language Hg. Though it is an algorithmic language in the sense of its approach to data manipulation and control flow, Hg is rather unique. The data structuring mechanism is based on hierarchical graphs, which give rise to very general data structures. This generality of data structures can cause problems in understanding the behavior of Hg programs, as a result of the *aliasing* that can take place in h–graphs.

It is this author's experience that those areas of programming languages that are most resistant to good verification techniques are also those areas that are most difficult to understand fully. This is one reason provision of verification systems capable of handling aliasing is so important. By studying the problems involved in verifying language constructs we can gain insight into improvements or changes that may be necessary or desirable in those constructs. In the verification system we develop, aliasing introduces addi–

tional complexity to program proofs, enough so as to suggest that aliasing is of questionable benefit in programming languages.

Why then, do so many languages permit aliasing to take place? Many features of modern languages can introduce aliasing. In block structured languages reference parameters in procedure calls can alias global variables. Reference parameters can be repeated in procedure calls. Accesses to arrays through subscript expressions can produce aliases of array elements. Pointer variables can access the same locations in memory. Renaming or equivalencing of storage is provided by FORTRAN as well as Ada.

Language designers must feel that the cost of alias introduction is warranted because the benefits of aliasing outweigh the difficulties. The benefits to programmers provided by arrays cannot be disputed. Reference parameters provide a cheap method of transferring results of procedure calls back to the calling scope. In addition, aliasing permits one to impose multiple viewpoints on data structures, isolating, for example, a deeply buried element of a data structure and giving it, for a short while, a new name of its own. But any language that permits programmer management of dynamic data structures with some pointer facility presents the programmer with the potential to create arbitrarily many aliases.

Given that aliasing is permitted in Hg, what can be done to keep it in check, thereby simplifying the verification of Hg programs? We tackle this problem by looking at data typing in Hg. We formalize the concepts of *data type* and *type correctness*, and show how the use of data types can structure our patterns of aliasing to permit easier verification of programs involving aliases.

Previous Studies of Aliasing

Most studies of program verification have ruled out aliasing in one form or another [7,8,9]. Although these works permit subscripted variables, which may generate array element aliases, they have ignored pointer variables and dynamic data structures.

Some previous works have supported verifying programs with some other form of aliasing. Cartwright and Oppen [1] present a method of handling a class of aliasing in programs. Their primary interest is providing the ability to verify programs with aliased parameters in a Pascal–like language. Arbitrarily complex dynamic data structures cannot be developed in this language. Although their method is sound and complete for the language considered these restrictions make their results inapplicable to practical languages such as Pascal. Olderog has presented sound and complete rules for Algol 60 with variables restricted to integer type [2]. This proof system, which is based on a copy–rule semantics for procedure calls, can be used to develop proofs for procedure calls which involve aliased parameters.

Major Results of This Research

The results of this work are presented in Chapters 2 through 5, with a detailed summary and evaluation in Chapter 6. We summarize the major ideas concisely here. After reviewing basic concepts related to h–graphs, we provide a concise and complete semantic definition of the language Hg. Hg is a powerful language which uses h–graphs as its data structuring mechanism. The language supports the development of general data structures involving pointers, typical control structures, a single assignment statement for atomic or structured values, procedure calls with value–result parameter transmission, and dynamic storage allocation.

We define an assertion language in which arguments about Hg program states can be made, and develop an axiom of assignment for Hg which completely captures the behavior of Hg assignments, even in the presence of complicated aliasing patterns. We also develop rules of inference so that one may develop formal proofs of entire Hg programs. We show that subject to several restrictions on procedure calls, the assignment axiom together with the inference rules form a *sound* and *relatively complete* Hoare style axiom system for the language Hg.

We develop a formal system of data types for h–graphs based on the idea of using h–graph grammars to specify program data types [6]. We incorporate this type system into the semantics of Hg, and formally define the concept of *type correctness* of a program. We then define the idea of a formal proof with respect to an h–graph grammar and show how by using this concept we can simplify proofs of Hg programs exhibiting certain data type characteristics.

Organization

The rest of this chapter is concerned with reviewing what is required to present a reasonable programming language specification and verification technique and noting some of the notation used in this work. In Chapter 2, an introduction to h–graphs is presented, followed by a brief review of some mathematical logic and a definition of the language Hg. The third chapter presents the assignment axiom for our verification system. Chapter 4 introduces the proof rules for Hg and shows that a system based on the assignment axiom and these rules is sound and relatively complete. We formalize the concept of data type as it pertains to h–graphs in Chapter 5. We also develop the idea of a proof of a formula with respect to a grammar and show how data type information can be used to simplify the verification of Hg programs under this proof system. We summarize the results of this work in Chapter 6.

1.1. Programs and Meaning

What does it mean to "verify a program?" When a program in the language of interest is executed, something will happen. The programmer probably has some notion of what the program should do, and what the program actually does. Verifying a program involves making sure these are one and the same. There are three principal tasks involved here :

(1) specifying what is expected of the program (program requirements),

(2) determining what the program does (program semantics), and

(3) assuring that these are equivalent (program proof).

Each of these tasks is considered briefly below; for a more complete introduction to the topic of program verification see [10,11].

Program Requirements

Many different properties might be used to dictate how a program is to behave. One might wish to constrain the time a program will take to execute on some particular machine, under what conditions it would halt, how much storage it would require on some machine, or what relation the final values of variables would have to the initial values. To this end, program requirement specification methods have been studied in many different works. Our intent here is not to shed any new light on how to specify requirements of programs. For our purposes, specification will be restricted to partial correctness assertions [12]. A *partial correctness assertion* consists of a program and two first order logical statements, one called the *precondition* and one the *postcondition.* The requirement specified by a partial correctness assertion is the following: The program, if it terminates at all when executed starting in a state satisfying the precondition, must terminate in a state satisfying the postcondition. An example of a partial correctness assertion is the following:

$$\{ x = 5 \}$$
$$x := x + 1$$
$$\{ x = 6 \}$$

The pre- and post-conditions are contained within the braces, the precondition preceding the statement, and the postcondition following the statement. This partial correctness assertion states that if the value of the variable x is 5, and the statement $x := x + 1$ is executed and terminates, then the value of x in the resulting state must be 6.

Two other kinds of assertions that are often discussed are *termination assertions* and *total correctness assertions.* A termination assertion asserts that a program will terminate. A total correctness assertion consists of a partial correctness assertion together with a termination assertion. Total correctness assertions are somewhat more restrictive than partial correctness assertions, and we lose no power by broadening our scope to look at programs that may not terminate.

Notice that these assertions must be *well formed formulas* of some symbolic logic. Subtleties may arise in the choice of this logic, and a more complete discussion of choosing an underlying logic is found in section 2.3.

Program Semantics

To determine the meaning of a particular program in a particular language, it is necessary to have a *semantic definition* of the programming language. There are many approaches to defining programming language semantics. Following Hoare and Lauer [13], we divide our discussion of semantic specification methods into two broad categories: constructive, or explicit, definitional methods and implicit methods.

The constructive approach to defining program semantics involves constructing the meanings of programs, either by specifying the effect of programs when executed on some virtual machine, or by constructing mathematical objects that represent the meanings of programs. Within the constructive methods, two broad categories of definitions can be isolated, *direct* and *operational* definitions.

By a *direct* definition, we mean a definition expressing only that information about the program state which is independent of any implementation. In particular there is no information on the state of any virtual machine defining the language. Machine independent information would include items such as order of statement execution and values of variables at given times in the execution. An *operational* definition, on the other hand, provides information that is not part of the program state. This information is typically

part of a virtual machine execution state, perhaps some auxiliary data structures or flags. The difference can be summed up by stating that operational definitions are those that provide some of the implementation details of a virtual machine, whereas direct ones do not.

Denotational definitions in the style of Scott and Strachey [14] are purely direct in nature. A denotational semantics for a language identifies a function with any given program in the language. This function is a map with an abstract state description as both its domain and range. There is no virtual machine to consider, so no implementation details are available.

Cook [15] uses a method of specification that is very nearly a direct one. He gives a proof system for a fragment of Algol 60. He defines the semantics of the language with a function *Comp* which, given a program and an initial state as its arguments, evaluates to the state sequence that the program generates when started in the initial state. Cook introduces a set of *registers* in order to make the function *Comp* conform to Algol scope rules. In doing so, he provides information that is not directly part of the program state.

As an example of the operational approach, the IBM Vienna Laboratory has used the Vienna Definition Language (VDL) [16] to define programming language semantics. Their method involves defining two components, a translator and interpreter, to give meanings to programs in the language. The translator translates from the target language into abstract syntax trees. The interpreter takes the tree for a program and the program's input and produces the output of the program. The state sequence of the interpreter for a given program defines that program's meaning. Interpretive models of semantics such as the well-known definition of LISP via an interpreter written in LISP [17] follow the work of McCarthy [18].

Pratt [19] has used Hierarchical Graph (H-graph) automata to specify language semantics. For each language, a string to graph grammar [20] specifies the translation from

program text into the data structure manipulated by the automaton. An H–graph automaton is defined that will accept this data structure as its input. The state sequence of this automaton defines the program semantics.

The unifying factor in the above described methods of language definition may not be obvious at first glance, but each of them gives a means of constructing the output state of a program given its input state. *Implicit* definitions do not provide this constructive framework. These definitions give information about programs from which one can deduce the properties of program states without constructing those states explicitly.

In the implicit approach, the semantics of a language is defined by stating properties of programs, but without providing an implementation model or mapping from the program to a function. Such a definition, being totally implementation independent, suffers as a language model for implementors, as it gives no hints about implementing complex constructs in the language. On the other hand, it is quite attractive to those who want to hide any implementation considerations and simply reason about the more abstract properties of the program.

A popular method of implicit specification has been to provide axioms and proof rules for the language of interest. This is the method introduced by Hoare [21]. The presumption is that the user, armed with this sort of definition, will be able to derive any properties of interest concerning his program. Portions of the language Pascal [9], and the language Euclid [8], among others, have been defined in this fashion.

Meyer and Halpern [22], in assessing this method of definition conclude that first–order partial correctness assertions of the type used can define the input–output semantics of program schemes from quite general programming languages, but they caution that the thesis that languages should be defined in this manner is "delicate" and "questionable." Several authors (e.g., [23,24]) note the subtle logical difficulties that arise in this style of definition.

Dijkstra's predicate transformers [25] are another implicit definitional technique. Dijkstra uses the weakest precondition predicate transformer, wp, to define the semantics of programs. $wp(K,Q)$ for some program segment K is defined to be that condition which is true of all states for which the program started in such a state will terminate in a state satisfying Q. Dijkstra includes termination of the program in his weakest precondition transformer, though this is not necessary. For deterministic programs predicate transformers are equivalent in power to partial–correctness assertions and termination assertions for defining language semantics. For nondeterministic programs, on the other hand, the methods are incomparable [26].

The approach taken to defining the language Hg is a direct constructive approach. The non–control–flow semantics of the language are defined in a denotational style, but control–flow semantics are defined in the style of Cook [15].

Demonstrating Correctness

We have seen that the partial correctness assertions may be used to specify program behavior requirements and that semantic definitions can be used to specify program behavior. How does one go about determining whether or not a given program in some language of interest satisfies some partial correctness assertions? One can, when presented with a proof system equipped to deal with partial correctness assertions, mathematically prove that the partial correctness assertions hold for a particular program.

A proof system of the sort we are alluding to generally consists of a set of axioms and inference rules. The axioms of the proof system are partial correctness assertions called "Hoare triples"

$$\{P\}K\{Q\}$$

where P and Q are *wff*'s (well–formed formulas) of the underlying logic and K is a statement of the language of interest. The inference rules are used to derive new true assertions from assertions already known to be true. Inference rules are generally written

$$\frac{s_1,\cdots,s_n}{s_{n+1}}$$

where each of $s_1,\cdots,s_{n+1}$ is either

(1) a wff of the underlying logic, or

(2) a Hoare triple.

The meaning of such an inference rule is that if the *premises* of the rule $s_1,\cdots,s_n$ are true wff's or Hoare triples, then the *conclusion* s_{n+1} is true as well. The Hoare triples cannot be formed into sentences in the logic by using sentential connectives such as *and*, *or*, etc. The special status of the Hoare triples in this scheme has drawn fire[24], but Greif and Meyer [26] have shown a method for incorporating Hoare triples into a logic with sentential connectives. If this is done a partial correctness semantics for a language can be expressed in terms of axioms alone without any inference rules.

There are two qualities we would like our proof system to embody. We would like our axioms and proof rules to be *sound*, meaning that anything deducible from our proof rules is true. We would also like our proof system to be *complete*, in the sense that if something is true of a program, we would like to be able to prove it. It is clear that the goal of completeness is doomed to failure, as many will recognize, since the formula $\{\,\mathbf{T}\,\}K\{\,\mathbf{F}\,\}$ is true only if K fails to halt, and this problem is known to be unsolvable. Cook [15] uses the notion of *relative* completeness of axiom systems to judge the completeness of a system without regard to this kind of inherent problem. A system is said to be *relatively complete*, or *complete in the sense of Cook* if, given that we are able to find the strongest postcondition $sp(K,P)$ for every statement K and precondition P, then our system is complete. Clarke has pointed out that this requirement is equivalent to the requirement that given that we are able to find the weakest precondition $wp(K,Q)$ for every statement K and postcondition Q our system is complete. In practice we may not be able to find the weakest precondition for a given statement and postcondition, but this is not the fault of our verification system, but the fault of the underlying logical system

on which our programming language is based.

Proof rules proposed for various languages have sometimes been shown unsound [24]. Merely having a set of axioms and proof rules does not guarantee that they represent a sound deductive system.

Since our language is defined constructively, we must determine a reasonable set of axioms and proof rules, and then demonstrate the soundness and relative completeness of our resulting deduction system. One prototype for such as task is found in [15], which provides an excellent presentation of the intricacies involved in developing good proof systems. Another important work demonstrating such a construction has been provided by de Bakker [27] wherein a simple programming language is defined denotationally and a Hoare style proof system for the language is developed.

Given that we have a sound and relatively complete proof system as outlined above, we may set about proving real programs. To prove an assertion about any particular program, say $\{P\}K\{Q\}$, one uses the axioms and proof rules as in any normal deductive system, until one has proved $\{P\}K\{Q\}$.

1.2. Notation

In this short section we present some basic notation that is used throughout the rest of this work.

We use the symbol $\mathbf{N}$ to denote the set of natural numbers. The whole numbers, $\mathbf{N} \sim \{0\}$ are denoted by $\mathbf{N}^+$.

Definition 1–1: (X^{N}, tuples over the set X)
$X^{\mathrm{N}} = \{\langle x_1, \cdots, x_n\rangle : \forall i, x_i \in X, n \in \mathbf{N}^+\}$. This is the set of all n–tuples over the set X, with typical element $\bar{x}$.

Definition 1–2: (X^{ω}, sequences over the set X)
$X^{\omega} = X^{\mathrm{N}} \cup \{\langle x_1, x_2, x_3, \cdots\rangle : \forall i, x_i \in X\}$. This is the set of all finite and infinite sequences over the set X.

Definition 1–3: ($\bar{x}\downarrow i$, ith element of a tuple or sequence)
If $\bar{x} \in X^{\omega}$ for some set X, $\bar{x}=\langle x_1,\cdots x_n\rangle$ or $\bar{x}=\langle x_1,x_2,\cdots\rangle$, $\bar{x}\downarrow i$ denotes the ith element of $\bar{x}$, that is x_i.

Definition 1–4: ($|\bar{x}|$, length of a tuple or sequence)
If $\bar{x} \in X^{\omega}$ and $\bar{x}=\langle x_1,x_2,\cdots\rangle$ then $|\bar{x}| = \infty$, otherwise if $\bar{x}=\langle x_1,\cdots,x_n\rangle$, then $|\bar{x}| = n$.

Definition 1–5: ($\bar{x}^{\frown}\bar{y}$, sequence concatenation)
Given sequences $\bar{x} = \langle x_1,\cdots,x_m\rangle$ and $\bar{y} = \langle y_1,\cdots,y_n\rangle$, the concatenation of $\bar{x}$ and $\bar{y}$, written $\bar{x}^{\frown}\bar{y}$ is the sequence $\langle x_1,\cdots,x_m,y_1,\cdots,y_n\rangle$.

Definition 1–6: ($A\hat{}\alpha$, singleton union)
Given a set A and an element α, $A\hat{}\alpha$ denotes the set $A \cup \{\alpha\}$.

Chapter 2

A Semantic Definition of the Language Hg

This work is an investigation into verifying certain properties of programs. We must agree on what language will be used to express these programs and on what programs in that language mean in order to proceed. The language we will be dealing with for the entirety of this work is the language Hg, which is based on hierarchical graphs (h–graphs). A syntactic and semantic definition of Hg is presented in this chapter. H–graphs provide the primary data types.

2.1. Data objects and selectors

H–graphs are simple structures if looked at intuitively, but some formalism is required to define them precisely. H–graphs are composed of hierarchies of directed graphs.

Atoms and Graphs

We assume as base sets, a set Φ of *nodes* and a set Ξ of *character values*.

Definition 2–1: (The Atoms, Δ)
An *atom* is a finite sequence of characters from Ξ. The set of all atoms, Δ, with typical element a, is defined to be the Kleene closure of Ξ, i.e., $\Delta=\Xi^*$ the set of all finite strings over Ξ. The atom # denotes the null (or empty) string.

We also assume the existence of an item $\perp$, called *undefined* or $\perp$, distinct from the elements of Φ and Δ.

Definition 2–2: (Graph)
An *extended directed graph* (or simply *graph*), g, over Φ and Δ is a triple $g=\langle M,E,s\rangle$ where
$M\subseteq\Phi$, M finite, nonempty (M is the *nodeset* of the graph g, denoted

nodeset(g)),
$E : M \times \Delta \rightarrow M$, E partial, finite (E is the *arcset*), and
$s \in M$ (s is the *initial node*).
If $E(m,a)=n$, then there is said to be "an arc labeled a from node m to node n."
A graph $g=\langle M,E,s \rangle$ is said to be *well–formed* iff

$$\forall m \in M,\ m \neq s,\ \exists a_1, \cdots, a_n,\ n>0,\ E(\cdots E(E(s,a_1),a_2)\cdots,a_n) = m,$$

in other words, there is a directed path from the initial node to any other node in the graph.

Definition 2–3: (Ω, the set of all graphs)
Ω is the set of all graphs with nodes from Φ, the universe of nodes, and arc labels from Δ the universe of atoms.

Graph Selectors

The ability to select a particular node from a graph is crucial to the development of a programming language, so we present here what are called *graph selectors*. A graph selector is a function that takes a graph as its argument and returns a node from that graph as its result.

The definition is given in two parts. First we define the syntax of graph selectors, *Gsel*; then we define the meaning of a graph selector $gsel \in Gsel$, denoted $[\![gsel]\!]$. The notation $[\![$ *syntactic entity* $]\!]$ is used when referring to the mathematical entity denoted by *syntactic entity*. This is necessary wherever we manipulate the syntax of an item in defining its meaning. In most works, specific domain mapping functions are presented to accomplish this task, but since the meaning of $[\![$ *syntactic entity* $]\!]$ is clear from its context in all uses here, naming these functions is unnecessary.

The productions used to define syntactic domains are presented in a somewhat informal way to avoid unnecessary complexity. One convention employed to aid readability is to use names of typical items of syntactic categories on the right hand side of productions rather than the names of the syntactic categories themselves. In keeping with this convention, we use the symbol a (and a_i) to represent an element of Δ in the following

definition since in definition 2.1 we stated that a is a typical element of Δ.

Definition 2–4: (*Gsel*, graph selectors)
The syntactic domain of graph selectors, *Gsel*, with typical element *gsel*, is defined as follows:

$gsel$ ::= /
| $/a$
| $/a_1.a_2$
| $/a_1.a_2.a_3$
...

Definition 2–5: ($[\![gsel]\!]$, the function denoted by $gsel \in Gsel$)

(a) The *arc traversal function* is $O:\Omega\times(\Phi\hat{}\perp)\times\Delta\rightarrow(\Phi\hat{}\perp)$, defined as follows:
If $g=\langle M,E,s\rangle\in\Omega, m\in M$, $a\in\Delta$, then

$$O(g,m,a)=\begin{cases} n \text{ iff } E(m,a)=n \\ \perp \text{ otherwise} \end{cases}$$

$$O(g,\perp,a)=\perp$$

(b) Let $g=\langle M,E,s\rangle\in\Omega$ and $a_1,\cdots,a_n\in\Delta$; $[\![gsel]\!]:\Omega\rightarrow\Phi\hat{}\perp$, the *graph selector function* denoted by the graph selector *gsel*, is defined by the following cases:
if $gsel \equiv /$ then $[\![gsel]\!](g) = s$ (/ is the *initial node selector*);
if $gsel \equiv /a_1$ then $[\![gsel]\!](g)=O(g, [\![/]\!](g), a_1)$;
if $gsel \equiv /a_1.\cdots.a_n$, $n>1$ then $[\![gsel]\!](g) = O(g, [\![/a_1.\cdots.a_{n-1}]\!](g), a_n)$.

(c) A node m of a graph g is *selectable* if there is a graph selector *gsel* such that $[\![gsel]\!](g)=m$.

In general we are interested only in the selectable nodes of a graph. Restricting ourselves to considering only well–formed graphs guarantees that all the nodes in any graph we look at are selectable, as the following result shows.

□

Theorem 2–1: If $g=\langle M,E,s\rangle$ is a graph and M_s is the set of nodes of M selectable in g, then $g_s=\langle M_s,E|M_s,s\rangle$ is a well formed graph.

Proof: Note first that g_s is indeed a graph. M_s is a finite nonempty subset of Φ, $E|M_s:M_s\times\Delta\rightarrow M_s$ since any node reachable by an arc from a selectable node must

itself be selectable (by concatenating the arc label onto the selector selecting the first node), and $s \in M_s$ since it is selectable with the selector /.
In addition a simple inductive argument shows that any node selectable in g is selectable in g_s with the same selector, and since each node in M_s is selectable in g it is also selectable in g_s. Since a selector $/a_1.\cdots.a_n$ simply defines a path from the initial node to the selected node, using arcs labeled $a_1, \cdots, a_n$, the existence of a selector for a node n guarantees the existence of a path to n from the initial node. Thus g_s must be well–formed.

□

A pictorial representation of a particular graph is given in Figure 2.1. The nodes of the graph are represented by circles, the arcs by labeled arrows, and the initial node is distinguished by an asterisk. Figure 2.1 also illustrates several selectors.

H–graphs

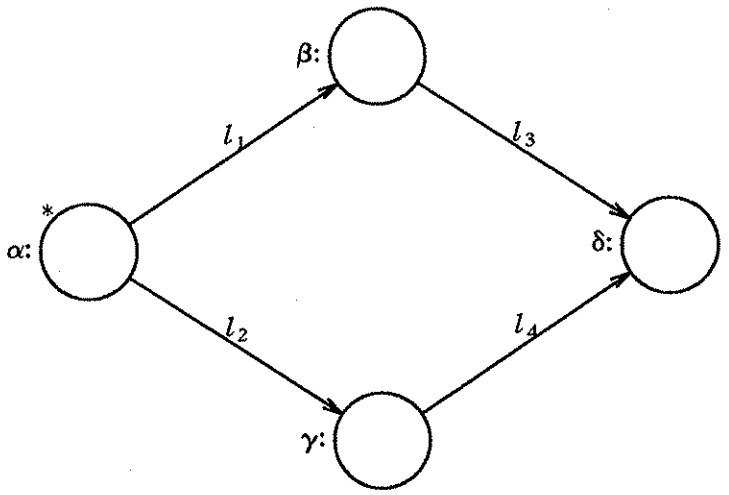
An h–graph consists of a set of graphs, a value function, and a distinguished root graph. The value function maps each node in a graph into a value. These values are either atoms in the atomset Δ or graphs in the graphset Ω. Because the value of a node in a graph may itself be a graph, graphs may be hierarchically nested, hence the name hierarchical–graph.

Definition 2–6: (h–graph)
An h–graph is a triple, $h=\langle G,V,r\rangle$ where
$G=\{g_1,\cdots,g_k\}, k \geq 1$, each $g_i=\langle M_i,E_i,s_i\rangle$, a well–formed graph over Φ and Δ, and $\forall i,j,\ i\neq j,\ M_i \cap M_j = \varnothing$ (G is the *graphset* of h),
$V:\bigcup_{i=1}^{k} M_i \rightarrow G \cup \Delta \hat{} \perp$, ($V$ is the *contents* or *value function*), and
$r \in G$ is the *root graph*.
For convenience, we define $V(\perp) = \perp$ in any h–graph. The set of all h–graphs is denoted Γ. The nodeset of the h–graph h, $nodeset(h) = \bigcup_{g\in G} nodeset(g)$.

The graph

$$g = \langle \{\alpha,\beta,\gamma,\delta\},\ \{\langle\alpha,l_1,\beta\rangle,\langle\alpha,l_2,\gamma\rangle,\langle\beta,l_3,\delta\rangle,\langle\gamma,l_4,\delta\rangle\},\ \alpha \rangle$$

is represented by the following diagram:



This graph consists of nodes α, β, γ, and δ connected by arcs labeled l_1, l_2, l_3, and l_4.

Some of the graph selectors of interest for this graph might be /, $/l_1$, $/l_1.l_3$, $/l_2.l_4$, and $/l_1.l_2$.
The selector function $[\![/]\!](g)$ returns the node α.
Selector function $[\![/l_1]\!](g)$ returns the node β.
Both $[\![/l_1.l_3]\!](g)$ and $[\![/l_2.l_4]\!](g)$ return the node δ.
Selector function $[\![/l_1.l_2]\!]$ returns $\perp$.

Figure 2.1 Example graph.

Definition 2–7: (V^+, the extended value function)
Let $h=\langle G,V,r\rangle$ be an h–graph. The *extended value function* V^+ for h, V^+:*nodeset*$(h)\rightarrow\Gamma\cup\Delta\hat{}\perp$, is defined as follows:

$V^+(n) = V(n)$ if $V(n)\in\Delta\hat{}\perp$.

$V^+(n) = h'$ if $V(n)\in\Omega$, where $h' = \langle G',V',V(n)\rangle$ with

G' defined recursively as follows:

$V(n)\in G'$

If $g\in G'$ and $m\in$ *nodeset*(g) such that $V(m)\in G$, then $V(m)\in G'$.

$V' = V|$*nodeset*(h')

Whenever $V^+(n)\in\Gamma$, $V^+(n)$ is termed the *sub–h–graph defined by node n.*

H–graph Selectors

Just as graphs have graph selectors, so do h–graphs have h–graph selectors. An h–graph selector is a function from an h–graph to a node within that h–graph. The selector is specified as the concatenation of graph selectors. To find the node selected, one starts at the root graph and applies the first graph selector. This returns a node. One then applies the contents function to that node to get another graph value, then applies the next graph selector to that graph, and so forth.

Definition 2–8: (*Sel*, h–graph selectors)
The syntactic domain of h–graph selectors, *Sel* with typical element s is defined as follows:

s ::= $gsel$
| $gsel_1 gsel_2$
...

where each of the $gsel_i \in Gsel$.

Definition 2–9: ($[\![s]\!]$, the function denoted by $s \in Sel$)
The function denoted by s in *Sel*, $[\![s]\!]: \Gamma \rightarrow \Phi \hat{} \perp$ is defined by the following cases:
Let $gsel_1, \cdots, gsel_k$ be graph selectors and let $h = \langle G, V, r \rangle \in \Gamma$.

if $s \equiv gsel_1$ then $[\![s]\!](h) = [\![gsel_1]\!](r)$

if $s \equiv gsel_1 \cdots gsel_k$, $k > 1$ then

$[\![s]\!](h) = [\![gsel_k]\!](V([\![gsel_1 \cdots gsel_{k-1}]\!](h)))$

if $V([\![gsel_1 \cdots gsel_{k-1}]\!](h)) \in G$; $\perp$ otherwise.

The following two results are of some interest and are easily demonstrated. Their proof is left to the reader.

Lemma 2–2: If $[\![gsel_1 \cdots gsel_k]\!](h) \neq \perp$, then for $1 < i \leqslant k$,
$[\![gsel_1 \cdots gsel_k]\!](h) = [\![gsel_i \cdots gsel_k]\!](V^+([\![gsel_1 \cdots gsel_{i-1}]\!](h)))$.

Proof: Left to the reader.

□

Lemma 2–3: $[\![s]\!](h) \neq \perp \Rightarrow [\![s]\!](h) \in nodeset(h)$

Proof: Left to the reader

□

Figure 2.2 shows a pictorial representation of an example h–graph and some h–graph selectors.

As with graphs, a node m in an h–graph h is said to be *selectable* if there is an h–graph selector s such that $[\![s]\!](h)=m$. An h–graph may have parts that are not selectable, but these parts are always complete graphs. That is, if the initial node of a graph is selectable, then every other node in that graph is selectable.

2.2. Modeling Data Objects Using H–graphs

In this section we give a convenient syntax for defining h–graphs and illustrate the modeling of various typical Pascal data objects using h–graphs.

A Syntax for H–graphs

For this work, we adopt a simplified form of the syntax for h–graphs used in the HOST programming environment [6]. The syntax is defined in Figure 2.3. Figure 2.4 shows the h–graph of Figure 2.2 written using this syntax.

Examples of Data Object Models

H–graphs provide straightforward formal models for most of the data objects that can be constructed in conventional programming languages such as Pascal. Here we show several Pascal program segments and an h–graph model of the data object constructed by each segment. In each case the variables declared are presumed to be part of a larger "activation record" data object called *local–state*, modeled as a single h–graph containing all the declared local variables of the Pascal procedure (or other program unit). Of course, only that part of the *local–state* relevant to each example is shown.

(a) **Simple variables**

The h–graph

$$h = \langle \{g_1,g_2,g_3\}, V_h, g_1 \rangle$$

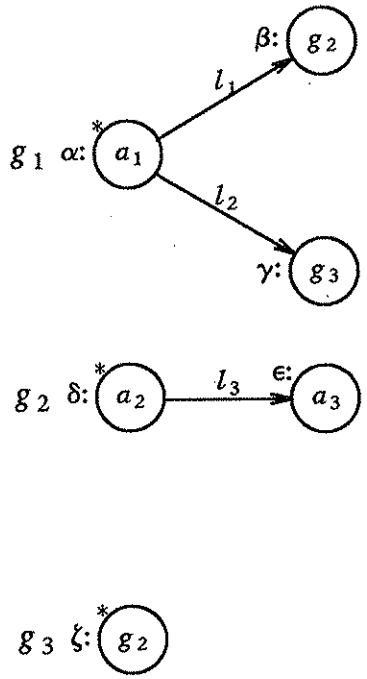
$$g_1 = \langle \{\alpha,\beta,\gamma\}, \{\langle \alpha,l_1,\beta \rangle, \langle \alpha,l_2,\gamma \rangle\}, \alpha \rangle$$

$$g_2 = \langle \{\delta,\epsilon\}, \{\langle \delta,l_3,\epsilon \rangle\}, \delta \rangle$$

$$g_3 = \langle \{\zeta\}, \varnothing, \zeta \rangle$$

$$V_h = \{\langle \alpha,a_1 \rangle, \langle \beta,g_2 \rangle, \langle \gamma,g_3 \rangle, \langle \delta,a_2 \rangle, \langle \epsilon,a_3 \rangle, \langle \zeta,g_2 \rangle\}$$

is represented by the following diagram:



This h–graph consists of the graphs g_1, g_2, and g_3. The root graph is g_1. Note that graph g_2 is the value of both node β and node ζ.

Selector function $[\![/l_2]\!](h)$ returns the node γ.
Selector function $[\![/l_2/]\!](h)$ returns the node ζ.
Selector function $[\![/l_2//]\!](h)$ returns the node δ.
Selector function $[\![/l_1/]\!](h)$ returns the node δ.
Selector function $[\![/l_1//]\!](h)$ returns $\perp$.

Figure 2.2 Example h–graph.

```
h-graph ::= h-graph-name : { root-graph
                             graph
                             ...
                             graph }

root-graph ::= graph

graph ::= graph-name : initial-node-arcset
                       node-arcset
                       ...
                       node-arcset

initial-node-arcset ::= node-arcset

node-arcset ::= node-name : [ node-value ]
                            - arc-label -> node-name
                            ...
                            - arc-label -> node-name

node-value ::= graph-name | atom

arc-label ::= atom

atom ::= character string

h-graph-name ::= identifier

graph-name ::= identifier

node-name ::= identifier
```

Figure 2.3 Syntax for h-graphs.

```
h: { g1: α: [ a1 ]
         - l1 -> β
         - l2 -> γ
     β: [ g2 ]
     γ: [ g3 ]

   g2: δ: [ a2 ]
         - l3 -> ϵ
     ϵ: [ a3 ]

   g3: ζ: [ g2 ]
 }
```

Figure 2.4 Figure 2.2 written in the new syntax.

The Pascal segment

```
var x : real;
 ...
begin
  x := 27.5
 ...
```

results in a local state modeled by the h–graph:

```
local–state : { g1: n1: [ # ]
                        - x -> n2
                n2: [ 27.5 ]
              }
```

The variable reference x is modeled by the h–graph selector /x applied to h–graph local–state.

(b) **Records**

The Pascal segment :

```
var R : record
          f1, f2 : integer;
        end;
 ...
begin
  R.f1 := 17;
  R.f2 := 18;
 ...
```

results in a local state modeled by the h–graph:

local–state: { g_1: n_1:[#]
– R –> n_2
n_2:[g_2]
g_2: n_3:[#]
– f1 –> n_4
– f2 –> n_5
n_4:[17]
n_5:[18]
}

The field reference R.f2 in Pascal is modeled by the h–graph selector /R/f2 applied to the h–graph local–state.

(c) **Pointers**

The Pascal segment :

```
type P1 = ↑integer ;
var z : P1;
 ...
begin
  new(z);
  z↑ := 17;
 ...
```

results in a local state modeled by the h–graph

local–state: { g_1: n_1:[#]
– z –> n_2
n_2:[g_2]
g_2: n_3:[17]
}

The Pascal pointer reference z↑ is modeled by the h–graph selector /z/ applied to the h–graph local–state.

(d) **Circular lists**

The Pascal segment

```
type link = ↑cell;
     cell = record
                    head : real;
                    tail : link;
            end;
var list : link;
 ...
begin
  new(list);
  new(list↑.tail);
  list↑.tail↑.tail) := list
  list↑.head := 22.2;
  list↑.tail↑.head := 33.3;
   ...
```

which generates a two-cell circular list as shown in Figure 2.5 results in a local state that is modeled by the h-graph

local-state: { g_1: n_1:[#]
- list -> n_2
n_2:[g_2]
g_2: n_3:[#]
- head -> n_4
- tail -> n_5
n_4:[22.2]
n_5:[g_3]
g_3: n_6:[#]
- head -> n_7
- tail -> n_8
n_7:[33.3]
n_8:[g_2]
}

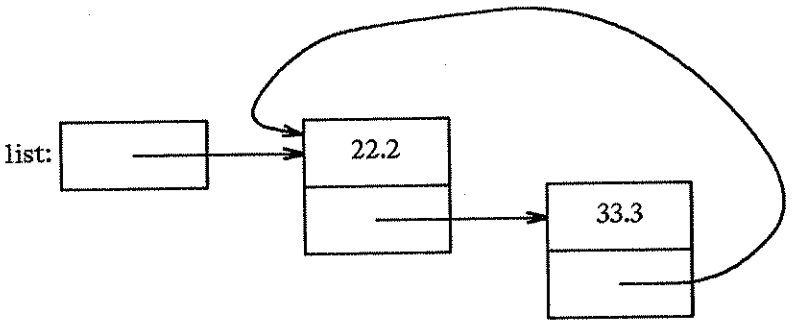


Figure 2.5 Circular list example.

The Pascal references and h–graph selectors correspond as follows:

Pascal	**h–graph**
list	*/list*
list↑.head	*/list /head*
list↑.tail↑.head	*/list /tail /head*
list↑.tail↑.tail	*/list /tail /tail*
list↑.tail↑.tail↑.head or list↑.head	*/list /tail /tail /head* or */list /head*

2.3. A Logical Model for Computation

Underlying any programming language there must be some set of values that forms the domain of the programs. A program can manipulate these values with some known functions, can determine the order of application of these functions dependent on the truth of predicates defined on these values, and can manipulate expressions involving constants drawn from this set of values.

In a typical programming language, one may also evaluate expressions involving values of variables and assign values to variables. As is shown in the next section, the language Hg has no variables per se. Their role is played by h–graph selectors in Hg. The state of an Hg program is not given as a function from variables into values, but as an h–graph.

Apart from their role as base elements in forming statements in Hg, the functions, predicates, constants, and h–graph selectors are used to make *assertions* about the state of an Hg program. It is these assertions that are used in Chapter 3 to construct a means of verifying properties of Hg programs. Thus the functions, predicates, constants, and selectors form the essential link between the Hg program itself and the mathematical logic (here termed the *underlying logic*) within which proofs about the program may be constructed.

The definitions that follow define the domains we use for predicate, constant, and function symbols.

Definition 2–10: (*Pred*, the predicate symbols)

The set *Pred*, with typical element ρ, is the set of *predicate symbols* of the underlying logic.

Definition 2–11: (*Const*, the constant symbols)
The set *Const*, with typical element c, is the set of *constant symbols* of the underlying logic.

Definition 2–12: (*Func*, the function symbols)
The set *Func*, with typical element f, is the set of *function symbols* of the underlying logic.

The meaning of a program is dependent on the choice of the meanings of the function, predicate, and constant symbols. A mathematical object giving meanings to these symbols is called a *structure.* A complete description of structures and how they relate to first–order logic can be found in Enderton for example [28]. Any candidate for serving as a structure for Hg must satisfy certain requirements. Ordinarily a particular set of functions and predicates would be chosen for Hg (e.g., the usual arithmetic and relation operations). However, for this study, it does not matter which particular functions and predicates are chosen (within the requirements set out below); therefore we leave the particular set unspecified.

Definition 2–13: ($\mathcal{U}$, an underlying structure for Hg)
An *underlying structure* $\mathcal{U}$ for Hg is a tuple
$\mathcal{U} = \langle Func^{\mathcal{U}}, Pred^{\mathcal{U}}, \mathcal{T}, \mathcal{U}_c, \mathcal{U}_f, \mathcal{U}_\rho \rangle$
satisfying

$Func^{\mathcal{U}} = \{f_1^{\mathcal{U}}, \cdots, f_k^{\mathcal{U}}\}$, each $f_i^{\mathcal{U}}: (\Omega \cup \Delta\hat{}\perp)^n \rightarrow \Delta\hat{}\perp$ for some $n \geq 0$ (each $f_i^{\mathcal{U}}$ is an atom–valued function taking graphs or atoms as arguments). Constant null–ary functions, those with no arguments, are permitted. We require that these functions be *strict*, that is, if any argument in a function application is $\perp$, then the function value is $\perp$.

$Pred^{\mathcal{U}} = \{\rho_1^{\mathcal{U}}, \cdots, \rho_j^{\mathcal{U}}\}$, each $\rho_i^{\mathcal{U}}: (\Omega \cup \Delta\hat{}\perp)^n \rightarrow \mathcal{T}$, for some $n > 0$ (each $\rho_i^{\mathcal{U}}$ is a predicate whose arguments are graphs or atoms).

$\mathcal{T} = \{ \mathbf{T}, \mathbf{F}, \mathbf{U} \}$, the set of *truth values* (discussed below).

$\mathcal{U}_c: Const \rightarrow \Phi \cup \Omega \cup \Delta\hat{}\perp$, an onto function ($\mathcal{U}_c$ gives the *interpretation* of the constant symbols; since $\mathcal{U}_c$ is onto, there is a constant symbol representing each graph, node, and atom).

$\mathcal{U}_f: Func \rightarrow Func^{\mathcal{U}}$ ($\mathcal{U}_f$ gives the interpretation of each function symbol, i.e., the

function it represents).

$\mathcal{U}_\rho$:$Pred \rightarrow Pred^{\mathcal{U}}$ ($\mathcal{U}_\rho$ gives the interpretation of each predicate symbol, i.e., the predicate it represents).

For simplicity we avoid the use of the interpretation functions $\mathcal{U}_c$, $\mathcal{U}_f$, and $\mathcal{U}_\rho$ for constant, function, and predicate symbols and instead write $c^{\mathcal{U}}$, $f^{\mathcal{U}}$, and $\rho^{\mathcal{U}}$ to mean $\mathcal{U}_c(c)$, $\mathcal{U}_f(f)$, and $\mathcal{U}_\rho(\rho)$ in the following.

The set $\mathcal{T}$ is the set of truth values of a first order logic with equality. The structure $\mathcal{U}$ can be thought of as an interface between the realm of h-graphs and the realm of logic. It links the h-graphs to this logic by assigning to each predicate symbol a mapping from n-tuples of elements of $\Omega \cup \Delta \hat{} \perp$, the value set of h-graphs, into $\mathcal{T}$, the truth value set of the logic. One would naturally think of choosing the standard first-order logic with the sentential connectives $\wedge$, $\vee$, $\neg$, $\supset$, etc. In such a system, a formula F is said to be *valid* under the structure $\mathcal{U}$, written $\models_{\mathcal{U}} F$, if for all assignments of values to free variables, $F = \mathbf{T}$, interpreting the functions, predicates, and constant symbols according to the definition of $\mathcal{U}$.

The standard logical system admits of two truth values, **T** and **F**. Often, however, evaluable constructs in actual programs have no distinct values, that is, they are undefined. Such undefined values may result in our h-graph language from the application of a selector s to an h-graph on which s is undefined, or from the application of a function that returns the undefined value $\perp$. It does not seem appropriate to require that a predicate operating on an undefined value should return **T** or **F**, rather we need a three-value logic with truth values { **T**, **F**, **U** }. This third value **U** means simply that the truth value in question is neither true, **T**, nor false, **F**.

There are many precedents for such an approach. Both Dijkstra [25] and Gries [7] implicitly use a three valued logic in defining the semantics of programs with the weakest precondition operator *wp*. Each of these works uses the *conditional* operators **cand** and **cor** in logical sentences. The domain of these operators includes an undefined value, so it is clear that a three valued logic is being employed. Other works [21,29,30,15] have

ignored the question of undefined program behavior entirely.

As far as what three valued logic to employ, the system of Kleene [31,32] is well suited to the task. Kleene developed the system so he could describe in logic the behavior of partial recursive functions. Kleene uses the convention that a predicate's truth value is **U** whenever one of its arguments is $\perp$. This can occur when a partial function is applied to a value not in its domain. Kleene's motivation is identical to ours, and since the value $\perp$ serves the purpose of the undefined value in the h–graph system, we require of each predicate $\rho^{\mathcal{U}}$ in the structure $\mathcal{U}$ that if any of the arguments to $\rho^{\mathcal{U}}$ is $\perp$, then the result of application of $\rho^{\mathcal{U}}$ is **U**. Truth tables for the sentential connectives of Kleene's logic are presented in figure 2.6. A system different from Kleene's could be used, with the proviso that when restricted to the values **T** and **F** it identically matches the standard two–valued logic.

The reader should be aware that it is not in general required that a logic have an equivalence operator, but we require one for our purposes. The equivalence operator = of the underlying logic is not a predicate in the usual sense. Its meaning is fixed separate from the structure $\mathcal{U}$, so it may return **T** or **F** for an invocation involving $\perp$, i.e., one may test for the undefined value $\perp$.

As stated earlier, Hg programs do not contain variables, only h–graph selectors. For the purpose of defining Hg, no mention of variables of any kind is necessary. If one

p	$\neg p$	q	$p \wedge q$			$p \vee q$			$p \supset q$		
		p	**T**	**U**	**F**	**T**	**U**	**F**	**T**	**U**	**F**
T	**F**	**T**	**T**	**U**	**F**	**T**	**T**	**T**	**T**	**U**	**F**
U	**U**	**U**	**U**	**U**	**F**	**T**	**U**	**U**	**T**	**U**	**U**
F	**T**	**F**	**F**	**F**	**F**	**T**	**U**	**F**	**T**	**T**	**T**

Figure 2.6 Truth Tables for Kleene's Three–Valued Logic.

wants to make assertions about Hg programs, though, one needs to be able to use variables for two important purposes.

First, one needs variables as place holders for values of interest. For example, one might want to say of some statement k that "if the selector $/a$ selects a node with value x before k executes then $/a$ will still have value x after k terminates." This use of the variable x is *universally quantified* since the statement must hold for all values x could take from $\Omega \cup \Delta \hat{} \perp$.

Secondly, one needs to be able to make assertions such as, "there is some value x such that selector $/a$ selects a node with value $f(x)$ and selector $/b$ selects a node with value $g(x)$." This use of the variable x is *existentially quantified* since there must *exist* a value for x which will make the statement hold.

A *free* variable, in logical nomenclature, is one which is not quantified. We use the convention that all free variables in logical formulae are implicitly universally quantified. Hence there will actually be no free variables. This deviates from the standard approach to program verification, in which free variables play the role of program variables, and the state of a program is captured by the variable assignment function of the underlying logic. We separate the program state from the underlying logic, thereby requiring less of the logic. The truth of an assertion will be determined solely on whether or not it is a tautology (i.e., true for all possible assignments of values to free variables, or in our special circumstances simply true). Logical variables are represented by the symbols x and y in the following.

2.4. Definition of the Language Hg

This section presents a syntactic definition of the language Hg and a direct constructive definition of its semantics, using the concepts developed in the previous section.

States and Variants of States

The state of the data at any point in the computation of an Hg program is described by an h–graph. This h–graph contains all of the nodes selectable by selectors in the statements of the program. Its value function maps each of these nodes into its value at that point of the computation.

Definition 2–14: (*Stat*, states of a computation)
Stat, the set of all states, with typical element σ, consists of the set of all h–graphs $h=\langle G,V,r\rangle$ such that for all $s\in Sel, s \equiv gsel_1$ if $[\![s]\!](\sigma)=n, n\in nodeset(r)$, then if $u\in Sel, u \equiv gsel_2, u\neq s, [\![u]\!](\sigma)\neq[\![s]\!](\sigma)$, together with the undefined state, $\perp_\sigma = \langle\emptyset,V_\perp,\perp\rangle$. $V_\perp$ is the constant function returning $\perp$. Where the context is clear, $\perp_\sigma$ will be written $\perp$.

The states are all those h–graphs satisfying the following condition: no h–graph selector consisting of a single graph selector selects the same node as some other single graph selector.

In defining a computational model for Hg programs, several sorts of mathematical substitution will be used. The next three definitions introduce the key concepts *value substitution, variant of state,* and *node replacement.* These concepts are used fairly heavily, so it is suggested the reader understand them fully.

Definition 2–15: ($f\{v:n\}$, value substitution)
Let $m,n\in domain(f^u)$, and $v\in range(f^u)$, then the *value substitution* of v for the value of n in the function f, written $f\{v:n\}$, is:

$$f\{v:n\}^u(m)=\begin{cases} f^u(m), & \text{if } m\neq n \\ v, & \text{if } m=n \end{cases}$$

We define the extension of value substitution to tuples in an obvious fashion. $f\{\bar{v}:\bar{n}\}$ where $|\bar{n}| = |\bar{v}|=m$ and $\forall i, 1\leq i\leq m, \bar{v}\downarrow i\in range(f^u)$, and $\bar{n}\downarrow i\in domain(f^u)$, is defined as follows:

$$f\{\bar{v}:\bar{n}\} = \begin{cases} f\{\bar{v}\downarrow 1:\bar{n}\downarrow 1\}\{\langle\bar{v}\downarrow 2,\cdots,\bar{v}\downarrow m\rangle:\langle\bar{n}\downarrow 2,\cdots,\bar{n}\downarrow m\rangle\} & \text{if } m>1 \\ f\{\bar{v}\downarrow 1:\bar{n}\downarrow 1\} & \text{if } m=1 \end{cases}$$

In other words, it is the result of substituting each pair of the vector elements in order from left to right.

Definition 2–16: ($\sigma\{v{:}s\}$, variant of state)
Given $\sigma=\langle G,V,r\rangle \in Stat$, $v \in (G \cup \Delta \hat{} \perp)$, $s \in Sel$, the *variant* of state $\sigma\{v{:}s\}$ denotes the state, $\sigma'=\langle G,V\{v{:}[\![s]\!](\sigma)\},r\rangle$ if $[\![s]\!](\sigma)\neq\perp$, and $\perp$ otherwise.

We define the extension of a state variant to vectors in the same way as value substitution, that is $\sigma\{\bar{v}{:}\bar{s}\}$ where $\sigma=\langle G,V,r\rangle$, $|\bar{v}|=|\bar{s}|=m$, and $\forall i, 1\leqslant i\leqslant m$, $\bar{v}\downarrow i \in (G \cup \Delta \hat{} \perp)$, $\bar{s}\downarrow i \in Sel$ is defined as follows:

$$\sigma\{\bar{v}{:}\bar{s}\} = \begin{cases} \sigma\{\bar{v}\downarrow 1{:}\bar{s}\downarrow 1\}\{\langle \bar{v}\downarrow 2,\cdots,\bar{v}\downarrow m\rangle{:}\langle \bar{s}\downarrow 2,\cdots,\bar{s}\downarrow m\rangle\} & \text{if } m>1 \\ \sigma\{\bar{v}\downarrow 1{:}\bar{s}\downarrow 1\} & \text{if } m=1 \end{cases}$$

Definition 2–17: (E_n^m, node replacement)
A node replacement for a node, tuple, set, or function E, written E_n^m, for nodes n and m is defined recursively as follows :

If E is a node, then $E_n^m = \begin{cases} m, & \text{if } E=n \\ E, & \text{otherwise} \end{cases}$
If E is a tuple, $\langle E_1,\cdots,E_k\rangle$ then E_n^m is $\langle E_{1n}^m,\cdots,E_{kn}^m\rangle$.
If E is a set, $\{E_1,\cdots,E_k\}$ then E_n^m is $\{E_{1n}^m,\cdots,E_{kn}^m\}$.
If E is a function, treat E as a set of ordered tuples in order to determine the value of E_n^m.
The expression E_n^m has the value E if E is not a node, tuple, set, or function.

The extension of node replacement to vectors, $E_{\bar{n}}^{\bar{m}}$, where $\bar{m}$ and $\bar{n}$ are identical length vectors of nodes and the $\bar{n}\downarrow i$'s are distinct is defined to be the *simultaneous* node replacement for each $n\downarrow i$ by the corresponding $m\downarrow i$ in E.

Expressions and Their Meanings

Functions operating on values selected by selectors may be composed to form more complex expressions. These expressions are used in assignment statements in the language Hg.

Definition 2–18: ($Expr$, the expressions)
The syntactic domain of expressions, $Expr$, with typical element, e, is defined by
$e ::= s$

$| f(e_1,\cdots,e_n)$

Only prefix expressions are used here in order to make the presentation as simple as possible.

The meaning of an expression is the mapping from states into values that it represents. In order to determine the value of an expression, one must evaluate the expression in the context of a particular data state. The function R defines the meaning of expressions. Given an expression in $Expr$, R maps it into a function from states, $Stat$, into values in $\Omega \cup \Delta \hat{} \perp$. Note that R does not map $Expr \times Stat$ into $\Omega \cup \Delta \hat{} \perp$, since the formulation here captures the idea that an expression has a fixed meaning but can result in different values in different states. We define R as follows:

Definition 2–19: (R, the expression value function)
$R:Expr \rightarrow (Stat \rightarrow \Omega \cup \Delta \hat{} \perp)$ is defined by the following cases:
$R(s)(\sigma)=V([\![s]\!](\sigma))$ where $\sigma=\langle G,V,r \rangle$.
$R(f(e_1,\cdots,e_n))(\sigma)=f^{u}(R(e_1)(\sigma),\cdots,R(e_n)(\sigma))$

In addition to expressions returning graph and atom values, we need to have expressions returning truth values. These expressions are used in the context of conditions for control structures in Hg programs. They are syntactically similar to expressions in $Expr$ but their meanings differ significantly so that they are not interchangeable with expressions in $Expr$.

Definition 2–20: ($Bool$, boolean expressions)
The domain of boolean expressions $Bool$, with typical element b, is defined by the following production:

$b ::= \rho(e_1,\cdots,e_n)$
$\quad | b_1 \wedge b_2$
$\quad | b_1 \vee b_2$
$\quad | \neg b$

Definition 2–21: (B, the boolean value function)
The meanings of elements of $Bool$ are given by the function $B:Bool \rightarrow (Stat \rightarrow \mathcal{T})$, defined recursively as follows:

$B(\rho(e_1,\cdots,e_n))(\sigma)=\rho^{\mu}(R(e_1)(\sigma),\cdots,R(e_n)(\sigma))$
$B(b_1\wedge b_2)(\sigma)=B(b_1)(\sigma)\wedge B(b_2)(\sigma)$
$B(b_1\vee b_2)(\sigma)=B(b_1)(\sigma)\vee B(b_2)(\sigma)$
$B(\neg b)(\sigma)=\neg B(b)(\sigma)$.

where the meanings of the predicates $\wedge$, $\vee$, and $\neg$ are given in Figure 2.6.

Statements

The statements of Hg are representative of the kinds of statements one finds in any modern programming language: assignments, control statements, and procedure calls. Procedure names are chosen from the atomset, Δ.

Definition 2–22: (*Pname*, procedure names)
The class of procedure names *Pname* with typical element p is defined as $Pname \subseteq \Delta$.

The next two (mutually recursive) definitions define the statements of the language Hg.

Definition 2–23: (*Stmt* the statements)
The statements *Stmt*, with typical element k, are defined by the following production: (The entries involving K refer to domain *Code* to be defined immediately following.) The empty statement is represented by ϵ.

k ::= ϵ
| $s := e$
| **if** b **then** K_1 **else** K_2 **endif**
| **while** b **loop** K **endloop**
| $p(s_1,\cdots,s_n)$

Definition 2–24: (*Code*, statement sequences)
The statement sequences *Code*, with typical element K, are defined by the following production:

K ::= k
| k ; K

The meanings of statements and statement sequences are defined by the function *Comp* in Definition 2–30 below.

Procedures and Programs

An Hg program consists of an initial state, a code segment, and a map from procedure names into their definitions. We first define the form of procedure definitions.

Definition 2–25: (*Pdef* , procedure definitions)
Pdef is the domain of procedure definitions. Each definition, $d \in Pdef$ is a triple

$$\langle \sigma, K, \bar{\eta} \rangle$$

where $\sigma = \langle G, V, r \rangle$ is an h–graph, the *initial state* of the procedure;
K is the *code* for the procedure;
and $\bar{\eta} \in Sel^N$ is a tuple of selectors, the *formal parameters* of the procedure, such that $\forall i, 1 \leqslant i \leqslant |\bar{\eta}|, [\![\bar{\eta} \downarrow i]\!](\sigma) \in nodeset(r)$, and
$\forall j, 1 \leqslant j \leqslant |\bar{\eta}|, i \neq j, [\![\bar{\eta} \downarrow i]\!](\sigma) \neq [\![\bar{\eta} \downarrow j]\!](\sigma)$

This definition restricts a procedure's formal parameters in two ways: they must select nodes in the rootgraph of the initial state, and no two may be aliased.

Definition 2–26: (*Pmap*, procedure name bindings)
If $\pi \in Pmap$, then $\pi: Pnames \rightarrow Pdef$ is a function from names of procedures into procedure definitions.

Definition 2–27: (*Prog*, Hg programs)
The class of programs *Prog* with typical element P consists of 3–tuples of the form $\langle \sigma, K, \pi \rangle$. where σ is the *initial state* of the program, K is the *code* for the program, and π defines the set of *procedure definitions* for the program, $range(\pi) = Pdef$, and $domain(\pi) = pnames(P)$, $pnames(P)$ is defined in the following definition.

Definition 2–28: ($pnames(P)$, procedure names of program P)
$pnames(P)$ for a program $P = \langle \sigma, K, \pi \rangle$ is defined recursively as follows:
(1) Every procedure name appearing in a procedure call in K is in $pnames(P)$.
(2) If $p \in pnames(P)$ and $\pi(p) = \langle \sigma_p, K_p, \bar{\eta}_p \rangle$ then every procedure name appearing in a procedure call in K_p is in $pnames(P)$.

In writing a program $\langle \sigma, K, \pi \rangle$ we use the following syntax:

program ::= **program**
σ < syntax from Figure 2.3 >
begin
K < syntax from definitions 2–23 and 2–24 >
end;

and each procedure definition $\langle \sigma_p, K_p, \overline{\eta}_p \rangle$ with its name p as given by π, is written:

procedure–defn ::= **procedure** p ($\overline{\eta}_p$);
σ_p <syntax from Figure 2.3 >
begin
K_p < syntax from definitions 2–23 and 2–24 >
end;

Note that we describe only those programs which have a static set of procedures. Procedures are not dynamically modifiable and there is no nesting of procedure definitions, hence there are none of the restrictions upon procedure definition which block structure might impose.

Semantics of Program Execution

The meaning of an Hg program is defined in terms of the sequence of states generated by execution of the program, where each state is an h–graph. The initial state is given directly by the program definition. Input data is assumed to be part of this initial state (since Hg contains no input–output statements). Similarly the results of execution are part of the final state, if such a state exists.

The function *Comp*, defined below, specifies how to generate the state sequence for any Hg program from a given initial state. The semantics for the usual control structures, composition, alternation (if), and iteration (while), is quite standard. Assignments and procedure calls have some subtleties that deserve mention.

Assignment

The right hand side of an assignment is an expression, which can take the form of a selector or a function call. If it is a function call, then the value assigned will necessarily be either an atom or $\perp$ since this function must be one of the base functions of the

underlying structure $\mathcal{U}$. If it is a selector, however, the value assigned is that contained in some node in the state h–graph, which may be an atom, a graph, or $\perp$.

It is the case of assignment of a graph–value from one node to another that lets us model pointers and data objects with pointer components but that introduces the problems of aliasing as well. The meaning of such a graph value assignment is simply that after execution the nodes selected by the left and right hand side selectors of the assignment both have the *same graph* as their values. Since this graph is part of the state before assignment, no new nodes are introduced by such an assignment. However, since the same graph value is accessible through both nodes, every node in the graph (and every node reachable through the nodes in the graph) now has an alias. Those nodes are now select–able through a selector which begins with the left–hand side selector and one which begins with the right–hand side selector.

For example, if a node α is selectable by selector $/s/b$ in a state σ in which no aliases currently exist, then after execution of the assignment

$$/t := /s$$

the node α must be selectable by $/t/b$ as well as $/s/b$ in the resulting state σ', so that $/t/b$ and $/s/b$ have become aliases. This is precisely what happens in a language like Pascal when a pointer variable is assigned the value of some other pointer variable.

Note that function calls in assignments in Hg have no side effects on the state, since these functions, though they may take graph–valued arguments, cannot return graph values.

Procedure call

The note in Definition 2–30 below explains in detail the steps involved in executing a procedure call: construction of the initial state for the procedure, insertion of this local state into the larger global state h–graph, assignment of actual parameter values to the formal parameter nodes, execution of the procedure body, and assignment of the result

values back to the actual parameter selectors. The assignment of arguments and result values has the same semantics as discussed above: graph values become the same between actual and formal parameter nodes. Formal parameter selectors must select nodes in the root graph of the procedure initial state and cannot be aliased with each other, but (as with languages such as Pascal) actual parameter selectors may be aliases. This leads to the usual problems of the sequence in which results are returned (assigned) to actual parameter selectors affecting the results (assignment to one of the actual parameters may destroy the result of a previous assignment through an alias). Results are returned in left–to–right order (see Definition 2–30).

Another subtlety involves side effects of procedure calls. If an actual parameter node has a graph value, then inside the procedure each node in this graph (and each node accessible through one of those nodes) is selectable using the corresponding formal parameter selector. Thus the procedure can use and modify the values of any of these nodes. Upon termination, after results are returned, all of these nodes retain their new values, and if they are still selectable in the calling state, then the new values are reflected there. Thus Hg procedure calls can modify any part of the calling state that is accessible through the actual parameters. However, if a node is not selectable through an actual parameter of a procedure call, that call **cannot** modify its value. Thus there are no hidden side effects of procedure calls, but every actual parameter is vulnerable to modification.

The final subtlety of procedure calls concerns what is commonly called *storage allocation.* New nodes are introduced into the global state by a procedure call, namely the nodes introduced in constructing the local state of the procedure. These new nodes represent the various local data objects of the procedure. At the time of procedure call, a copy of the local state h–graph is merged with the existing global state h–graph (the calling state) and the root graph of the local state becomes the root graph of the global state. This makes the local state nodes selectable via the selectors that appear in the procedure's

code. Upon termination, the root graph is changed back to that of the (calling) global state at the time of entry. Ordinarily this change of root graph makes the procedure's local state no longer accessible via selectors (i.e., these nodes become garbage). However, the procedure may assign part of its local state to a formal parameter selector during its execution. On return, that part of the local state remains visible, now selectable through the corresponding actual parameter selector. The result is that new data objects (nodes, atoms, and graphs) are now visible to the caller, exactly as if the called procedure had allocated storage, created these new data objects, and returned them to the caller. In this way *constructor* procedures, such as the Pascal *new* operation, may be modeled in Hg.

In giving the definition of *Comp*, the function that defines the semantics of Hg programs, we often need to designate the last state of a state sequence.

Definition 2–29: (*Last*, last element of a state sequence)
The function $Last:Stat^{\omega}\rightarrow Stat$ is defined for state sequence $\overline{\sigma}$ as follows:

$$Last(\overline{\sigma})=\begin{cases}\overline{\sigma}\downarrow n & \text{if } \overline{\sigma} \text{ is finite and } |\overline{\sigma}|=n\\ \perp & \text{otherwise}\end{cases}$$

Definition 2–30: ($Comp_P$, computation sequence function for program P)
$Comp_P:Code\rightarrow(Stat\rightarrow Stat^{\omega})$ is a function mapping statement sequences in a program into functions from states to state sequences. The state sequence function defined by application of $Comp_P$ to a program $P=\langle\sigma,K,\pi\rangle$ is defined below for each case of *Code*. Alternatives for *Code* appear with the corresponding value for $Comp_P(Code)(\sigma)$ to the right of the arrow, "→". For all values of *Code*, $Comp_P(Code)(\perp) = \langle\perp\rangle$.

Cases of *Code*:

$\epsilon \rightarrow \langle\sigma\rangle$

$s := e \rightarrow \langle\sigma\{R(e)(\sigma):s\}\rangle$

In other words, if the selector on the left hand side of an assignment is defined in the starting state, then in the resulting state the node it selects will contain the value of the expression e. If the left hand side selector is undefined, then the definition of value substitution guarantees that the resulting state is $\perp$.

$k;K \rightarrow Comp_{\not p}(k)(\sigma)^{\frown} Comp_{\not p}(K)(Last(Comp_{\not p}(k)(\sigma)))$

if b **then** K_1 **else** K_2 **endif** $\rightarrow$

$$\begin{cases} Comp_{\not p}(K_1)(\sigma) & \text{if } B(b)(\sigma) = \mathbf{T} \\ Comp_{\not p}(K_2)(\sigma) & \text{if } B(b)(\sigma) = \mathbf{F} \\ \langle \perp \rangle & \text{otherwise} \end{cases}$$

while b **loop** K **endloop** $\rightarrow$

$$\begin{cases} \langle \sigma \rangle^{\frown} Comp_{\not p}(K)(\sigma)^{\frown} Comp_{\not p}(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(Last(Comp_{\not p}(K)(\sigma))) \\ \quad \text{if } B(b)(\sigma) = \mathbf{T} \\ \langle \sigma \rangle \text{ if } B(b)(\sigma) = \mathbf{F} \\ \langle \perp \rangle \text{ otherwise} \end{cases}$$

$$p(\bar{\alpha}) \rightarrow \begin{cases} \langle \sigma', \sigma'' \rangle^{\frown} Comp_{\not p}(K_p)(\sigma'')^{\frown} \langle \sigma''' \rangle \\ \quad \text{if } \pi(p) = \langle \sigma_p, K_p, \bar{\eta}_p \rangle \text{ and } |\bar{\eta}_p| = |\bar{\alpha}| \\ \langle \perp \rangle \text{ otherwise} \end{cases}$$

where σ', σ'', and σ''' are defined below, and $\bar{\alpha} \in Sel^{\mathbf{N}}$.

The first alternative applies if the number of parameters in the call and definition of p match. Otherwise the result of the procedure call is $\perp$.

$\sigma' = \langle G', V', r' \rangle$ is constructed as follows:

Given that $\sigma_p = \langle G_p, V_p, r_p \rangle$, and $\sigma = \langle G, V, r \rangle$,

(1) form a tuple $\bar{n}$ from all of the nodes in $nodeset(\sigma_p)$.

(2) Choose a set of $|\bar{n}|$ nodes from $\Phi \sim (nodeset(\sigma) \cup nodeset(\sigma_p))$ and form a tuple $\bar{m}$ from these.

$G' = G \cup G_{p\frac{\bar{m}}{\bar{n}}}$, $V' = V \cup V_{p\frac{\bar{m}}{\bar{n}}}$, and $r' = r_{p\frac{\bar{m}}{\bar{n}}}$.

$\sigma'' = \sigma'\{R(\bar{\alpha})(\sigma) : \bar{\eta}_p\}$

$$\sigma''' = \begin{cases} \sigma^*\{R(\bar{\eta}_p)(Last(Comp_{\not p}(K_p)(\sigma''))) : \bar{\alpha}\} \\ \quad \text{if } Last(Comp_{\not p}(K_p)(\sigma'')) \neq \perp \\ \perp \quad \text{otherwise} \end{cases}$$

Where $\sigma^* = \langle G^*, V^*, r \rangle$, and $\langle G^*, V^*, r^* \rangle = Last(Comp_{\not p}(K_p)(\sigma''))$. That is, σ^* is a state identical to $Last(Comp_{\not p}(K_p)(\sigma''))$, but with the same rootgraph as the calling state.

Note: The sequence of states produced by a procedure call is divided here into four parts. First an initial state σ' for the procedure is created as a copy of the initial local state σ_p given in the procedure definition, using nodes currently not in use by the program. Note that σ' has a root different from σ. Secondly the actual parameters' values are assigned to the formal parameter locations in the

procedure's state. This results in the second state in the sequence σ''. Note that the only information common to the procedure and the rest of the program is that contained in the parameters. Thirdly the states produced by execution of the procedure, $Comp\,\mathcal{K}(K_p)(\sigma'')$, are included. And finally the formal parameters' final values are assigned back into the actual parameter locations, and the root graph is changed back to the rootgraph in the calling state σ, resulting in the final state σ'''.

During execution of a program, nodes are never deleted from the state, as the following theorem shows, although nodes may become inaccessible (i.e., garbage):

Theorem 2–4: For any state σ and code K, if $Last(Comp\,\mathcal{K}(K)(\sigma)) \neq \perp$ then $nodeset(\sigma) \subseteq nodeset(Last(Comp\,\mathcal{K}(K)(\sigma))$.

Proof: By the definition of *Comp*, if any state in $Comp\,\mathcal{K}(K)(\sigma) = \perp$, then all states following that are $\perp$. Hence we can assume for any state σ' in $Comp\,\mathcal{K}(K)(\sigma)$, $\sigma' \neq \perp$ holds. The proof proceeds by simultaneous induction on the cases of K.

ϵ:

$Last(Comp\,\mathcal{K}(\epsilon)(\sigma))=\sigma$ so clearly $nodeset(\sigma) \subseteq nodeset(Last(Comp\,\mathcal{K}(K)(\sigma))$.

$s := e$:

$Last(Comp\,\mathcal{K}(s := e)(\sigma))=\sigma\{R(e)(\sigma):s\}$, and by Definition 2–16,
$nodeset(\sigma) = nodeset(\sigma\{R(e)(\sigma):s\})$.

$k;K$:

By induction, assume $nodeset(\sigma) \subseteq nodeset(Last(Comp\,\mathcal{K}(k)(\sigma)))$
and
$nodeset(Last(Comp\,\mathcal{K}(k)(\sigma))) \subseteq nodeset(Last(Comp\,\mathcal{K}(K)(Last(Comp\,\mathcal{K}(k)(\sigma)))))$
therefore $nodeset(\sigma) \subseteq nodeset(Last(Comp\,\mathcal{K}(k;K)(\sigma)))$

if b **then** K_1 **else** K_2 **endif** :

If $B(b)(\sigma)=$ **T** in σ, then since by induction we assume

$$nodeset(\sigma) \subseteq nodeset(Last(Comp\,\mathcal{K}(K_1)(\sigma)))$$

we have our result.

If, on the other hand, $B(b)=$ **F**, we have the same result for

$$nodeset(Last(Comp\,\mathcal{K}(K_2)(\sigma))).$$

If neither of these conditions is satisfied then

$$Last(Comp\,\mathcal{K}(\textbf{if } b \textbf{ then } K_1 \textbf{ else } K_2 \textbf{ endif })(\sigma)) = \perp$$

which contradicts our hypothesis.

while b **loop** K **endloop** :

can be demonstrated in the same manner as for the **if** statement above.

$p(\bar{\alpha})$:

We show for σ', σ'', and σ''' as in *Comp*, that

(1) $nodeset(\sigma) \subseteq nodeset(\sigma')$,
(2) $nodeset(\sigma') = nodeset(\sigma'')$, and
(3) $nodeset(\sigma'') \subseteq nodeset(\sigma''')$
so our result will hold, since $Last(Comp\,\not(p(\overline{\alpha}))(\sigma)) = \sigma'''$.
(1) Suppose $\sigma = \langle G,V,r \rangle$ and $\sigma' = \langle G',V',r' \rangle$ as defined in *Comp*. Then

$$\begin{aligned} nodeset(\sigma) &= \bigcup_{g \in G} nodeset(g) \\ &\subseteq \bigcup_{g \in G'} nodeset(g) \\ &= nodeset(\sigma'). \end{aligned}$$

since $G \subset G'$ by the definition of *Comp*,
(2) Since $\sigma'' = \sigma'\{R(\overline{\alpha})(\sigma):\overline{\eta}\}$, and by the definition of variant of state, σ'' and σ' have identical graphsets, we have

$$nodeset(\sigma') = nodeset(\sigma'')$$

(3) By the inductive hypothesis, we assume
$nodeset(\sigma'') \subseteq nodeset(Last(Comp\,\not(K_p)(\sigma')))$ and since
$G(\sigma''') = G(Last(Comp\,\not(K_p)(\sigma')))$ we know that

$$nodeset(\sigma'') \subseteq nodeset(\sigma''')$$

□

Corollary 2–5: In Definition 2–30, procedure call case,
$nodeset(\sigma) \subseteq nodeset(\sigma') = nodeset(\sigma'') \subseteq nodeset(\sigma''')$

Proof: Demonstrated above.

□

It is also useful to know that the root graph of the state is invariant during program execution (except temporarily during a procedure call).

Theorem 2–6: If $\sigma=\langle G,V,r \rangle$ and $Last(Comp\,\not(K)(\sigma))=\langle G^*,V^*,r^* \rangle \neq \perp$ for arbitrary code K, then $r^*=r$.

Proof: Trivial induction on the cases of K.

□

Recall that the definition of *Stat* requires that there are no aliases between single graph selectors selecting nodes in the rootgraph. The definition of *Comp* ignores this condition; but Theorem 2–6 guarantees that if the condition holds in some state then

execution of any statement will preserve its truth.

Corollary 2–7: If $\sigma=\langle G,V,r\rangle \in Stat$ and $Last(Comp_P(K)(\sigma))=\langle G^*,V^*,r^*\rangle \neq \perp$ for arbitrary code K, then for all $s \in Sel, s \equiv gsel_1$ if $[\![s]\!](\sigma)=n, n \in nodeset(r)$, then if $u \in Sel, u \equiv gsel_2, u \neq s, [\![u]\!](\sigma) \neq [\![s]\!](\sigma)$, in other words $Last(Comp_P(K)(\sigma)) \in Stat$

□

When the program P is clear from the context, or unimportant to the discussion at hand, we simply write *Comp* instead of $Comp_P$.

2.5. Example Hg Programs

The following two examples show how Hg programs look. In each case we present a program and show the final state resulting from execution of that program. We make one slight extension to the syntax for h–graphs in Figure 2.3: if a node in a graph has no exiting arcs and only a single entering arc, we write its value immediately after the first appearance of its name, as for nodes n_2–n_5 below. The usual arithmetic and relational operations are used for the base functions and predicates of the underlying logic. Constants like 0 and 1 appearing in statements of Hg programs represent null–ary constant functions returning the value represented by the name of the function.

Example 2–1: (Division Program)

The procedure *divide* of the program below is the standard algorithm for integer division by repeated subtraction. The main program's initial state contains data values for the computation of 17 divided by 3.

```
program
prog-local-state : {  g1:n1:[ # ]
                        - a -> n2:[ 17 ]
                        - b -> n3:[ 3 ]
                        - quotient -> n4:[ 0 ]
                        - remainder -> n5:[ 0 ]
                   }
```

```
begin
   divide ( /a, /b, /quotient, /remainder )
end;

procedure divide ( /x, /y, /q, /r );

        divide-local-state : { g1:n1:[ # ]
                                    - x -> n2:[ 0 ]
                                    - y -> n3:[ 0 ]
                                    - q -> n4:[ 0 ]
                                    - r -> n5:[ 0 ]
                             }

begin
        /q := 0;
        /r := /x;
         while /r ≥ /y loop
                /r := /r - /y;
                /q := /q + 1;
         endloop
end;
```

Execution of the program above will result in the following h-graph final state: (Note that the copy, g_2, of the initial local state h-graph divide-local-state, constructed at the time of call of the divide procedure, remains as an inaccessible part of the final state h-graph.)

```
prog-local-state : {  g1:n1:[ # ]
                            - a -> n2:[ 17 ]
                            - b -> n3:[ 3 ]
                            - quotient -> n4:[ 5 ]
                            - remainder -> n5:[ 2 ]

                      g2:n6:[ # ]
                            - x -> n7:[ 17 ]
                            - y -> n8:[ 3 ]
                            - q -> n9:[ 5 ]
                            - r -> n10:[ 2 ]
                   }
```

Example 2-2: (Stack Program)

Procedure *push* of the program below is a simple procedure to insert a new element at the head of a singly linked list (stack). The main program provides data to construct a stack containing the two values 3 and 6 by pushing each value onto an initially empty

stack.

```
program
prog-local-state : {  g1:n1:[ # ]
                        - value1 -> n2:[ 6 ]
                        - value2 -> n3:[ 3 ]
                        - stack -> n4:[ nil ]
                   }

begin
        push( /value1, /stack );
        push( /value2, /stack )
end;

procedure push ( /value, /stk );

        push-local-state : {  g1:n1:[ # ]
                                - newelement -> n2:[ g2 ]
                                - value -> n3:[ 0 ]
                                - stk -> n4:[ nil ]

                              g2:n5:[ # ]
                                - head -> n6:[ 0 ]
                                - tail -> n7:[ nil ]
                           }

 begin
        /newelement/head := /value;
        /newelement/tail := /stk;
        /stk := /newelement
 end
```

Execution of this program results in the following h-graph final state:

```
local-state: {  g1:n1:[ # ]
                    - value1 -> n2:[ 6 ]
                    - value2 -> n3:[ 3 ]
                    - stack -> n4:[ g2 ]

                g2:n5:[ # ]
                    - head -> n6:[ 3 ]
                    - tail -> n7:[ g3 ]

                g3:n8:[ # ]
                    - head -> n9:[ 6 ]
                    - tail -> n10:[ nil ]

                g4:n11:[ # ]
                    - newelement -> n12:[ g3 ]
                    - value -> n13:[ 6 ]
                    - stk -> n14:[ g3 ]

                g5:n15:[ # ]
                    - newelement -> n16:[ g2 ]
                    - value -> n17:[ 3 ]
                    - stk -> n18:[ g2 ]
            }
```

Graphs g_4 and g_5 represent copies of the graph "push-local-state" remaining from the two calls of procedure *push*, now inaccessible garbage.

2.6. Chapter Summary

In Section 1, we present the definition of an h-graph and h-graph selector together with supporting definitions. We present a pictorial method of representing h-graphs and show examples of applying selectors to particular h-graphs. Section 2 introduces a more practical syntax for writing h-graphs and selectors and shows the modeling of typical data objects from Pascal. The role of an underlying logical model for a programming language is discussed in Section 3. We argue for using three-valued logic in programming language definition, citing precedents and giving an example of such a system. Section 4 brings together h-graphs and a three-valued logic in the definition of the programming language Hg. The state of a program is represented as an h-graph. Each statement of the language maps starting states into resulting states. The meaning of execution of a program in the language is defined by providing a function that generates the sequence of states

that would occur as a result of executing the statements of that program. Finally in Section 5, a set of example Hg programs is given and the final state resulting from their execution is specified.

Chapter 3

The Assignment Axiom

Now that we have a good idea of what Hg programs mean, we are faced with the question of formulating and verifying properties of particular programs. The properties of interest in this work are properties in first-order predicate logic. We have already assumed there is some underlying structure $\mathcal{U}$ upon which programs are built. It is now our task to provide a logic for arguing about the correctness of programs themselves.

We first extend the existing definitions to permit the expression of the properties of which we want to determine the correctness. We then present a method for determining what properties are true of assignment statements. Methods for proving properties of other statements in Hg are developed in Chapter 4.

3.1. Assertions and Correctness Formulae

We now define the syntax of assertions that are used for asserting facts about program states. Properties that cannot be expressed by this language lie outside of this proof system. The language of these assertions is quite rich though, so it is unlikely that one would need to express concepts not embodied therein.

One may make arguments about the values of selectors using any predicates and functions defined in the underlying structure $\mathcal{U}$. In addition one may state properties involving variables that are quantified over the set of values. We need to be able to reference selector values, variables, constants, and function values in assertions.

Definition 3-1: (*Term*, terms of assertions)
The syntactic class *Term* of *assertion terms*, with typical element d, is defined by the following production:

$d ::= s$
$\quad | \; x$
$\quad | \; c$
$\quad | \; f(d_1, \cdots, d_n)$

Definition 3–2: (*Assn*, the assertions)
The domain of assertions *Assn*, with typical elements P and Q, is defined by the following production:

$P ::= \exists x(P)$
$\quad | \; \neg P$
$\quad | \; \rho(d_1, \cdots, d_n)$
$\quad | \; P \wedge Q$
$\quad | \; P \vee Q$
$\quad | \; P \supset Q$
$\quad | \; d_1 = d_2$
$\quad | \; eq(s_1, s_2)$
$\quad | \; eq(s, x)$
$\quad | \; eq(\perp, x)$

We define the meaning of an assertion to be a function from program states into an assertion involving no selectors. When we introduce correctness formulae that involve these assertions, we define the truth of such a formula by analogy with a formula of the underlying logical structure $\mathcal{U}$ which is true in exactly the same states. Note that a formula so constructed must not reference selectors since there is no component of the underlying logic which corresponds to selectors. To remove these selectors from the assertions which appear in the formulae, we evaluate each selector in the state of interest and replace it with a constant which has the same value. This is done in the term value function *Tvalue*.

Definition 3–3: (*Tvalue*, the term value function)
The term value function $Tvalue: Term \rightarrow (Stat \rightarrow Term)$ maps terms in *Term* into functions from the states into terms in which no selectors appear. It is defined by the following cases:

$Tvalue(s)(\sigma) = c$ such that $c^{\mathcal{U}} = R(s)(\sigma)$
$Tvalue(x)(\sigma) = x$
$Tvalue(c)(\sigma) = c$
$Tvalue(f(d_1, \cdots, d_n))(\sigma) = f(Tvalue(d_1)(\sigma), \cdots, Tvalue(d_n)(\sigma))$

Definition 3–4: ($[\![P]\!]$, function denoted by the assertion P)
If $P \in Assn$, then $[\![P]\!]:Stat \rightarrow P$ maps assertions into functions from states into assertions in which no selectors appear. $[\![P]\!]$ is defined as follows:

$[\![\exists x(P)]\!](\sigma) = \exists x([\![P]\!](\sigma))$
$[\![\neg P]\!](\sigma) = \neg[\![P]\!](\sigma)$
$[\![\rho(d_1, \cdots, d_n)]\!](\sigma) = \rho(Tvalue(d_1)(\sigma), \cdots, Tvalue(d_n)(\sigma))$.
$[\![P \wedge Q]\!](\sigma) = [\![P]\!](\sigma) \wedge [\![Q]\!](\sigma)$
$[\![P \vee Q]\!](\sigma) = [\![P]\!](\sigma) \vee [\![Q]\!](\sigma)$
$[\![P \supset Q]\!](\sigma) = [\![P]\!](\sigma) \supset [\![Q]\!](\sigma)$
$[\![d_1 = d_2]\!](\sigma) = (Tvalue(d_1)(\sigma) = Tvalue(d_2)(\sigma))$
$[\![eq(s_1, s_2)]\!](\sigma) = (c_1 = c_2)$ such that $c_1{}^u = [\![s_1]\!](\sigma)$ and $c_2{}^u = [\![s_2]\!](\sigma)$
$[\![eq(s, x)]\!](\sigma) = (c = x)$ such that $c^u = [\![s]\!](\sigma)$
$[\![eq(\perp, x)]\!](\sigma) = (\perp = x)$

Definition 3–5: ($Form$, correctness formulae)
The domain of correctness formulae $Form$, with typical element F, is defined by the following production rule

$F ::= \{P\}$
$\quad | \{P\}K\{Q\}$
$\quad | F_1 \wedge F_2$

A formula $\{P\}K\{Q\}$ is known as a *Hoare triple*, with *precondition* P and *postcondition* Q.

Definition 3–6: ($[\![F]\!]$, function denoted by a correctness formula)
The function denoted by the correctness formula F, $[\![F]\!]:Stat \rightarrow \mathcal{T}$, is defined recursively as follows:

$[\![F]\!](\perp) = \mathbf{F}$

$$[\![\{P\}]\!](\sigma) = \begin{cases} \mathbf{T} & \text{if } \models_u [\![P]\!](\sigma) \\ \mathbf{F} & \text{otherwise} \end{cases}$$

$$[\![\{P\}K\{Q\}]\!](\sigma) = \begin{cases} \mathbf{T} & \text{if } \models_u ([\![P]\!](\sigma) \wedge |Comp(K)(\sigma)| \in \mathbf{N} \supset [\![Q]\!](Last(Comp(K)(\sigma)))) \\ \mathbf{F} & \text{otherwise} \end{cases}$$

$$[\![F_1 \wedge F2]\!](\sigma) = \begin{cases} \mathbf{T} & \text{if } \models_u ([\![F_1]\!](\sigma) \wedge [\![F_2]\!](\sigma)) \\ \mathbf{F} & \text{otherwise} \end{cases}$$

Note that the meaning of correctness formulae of the form $\{P\}K\{Q\}$, is that the truth of P in some starting state implies (or better *guarantees*) that either Q is true in the state resulting from execution of K, or that execution of K does not terminate. If $P \in Assn$, then the difference between $[\![P]\!](\sigma)$ and $[\![\{P\}]\!](\sigma)$ lies in the fact that $[\![P]\!](\sigma)$

is a wff in the underlying logic constructed by replacing selectors in P by constants, while $[\![\{P\}]\!](\sigma)$ is the truth value of the wff $[\![P]\!](\sigma)$ *when evaluated* in the underlying logic. For this work it is only the truth value $[\![\{P\}]\!](\sigma)$ which is of interest, not the structure of the formula $[\![P]\!](\sigma)$. Therefore in the following we ordinarily write $[\![P]\!](\sigma)$ to mean the truth value rather than the formula, as in the following definition.

The last three alternatives of Definition 3–4, involving the function *eq*, are tests for node identity. In other words, $[\![eq(s_1,s_2)]\!](\sigma)$ is true if the selectors s_1 and s_2 select the same node in the state σ. This condition captures the notion of an *alias* and is at the center of much of the work to follow.

Definition 3–7: (selectors aliased in a state)
Two distinct selectors s_1 and s_2 are said to be *aliased* in the state σ, if $[\![eq(s_1,s_2)]\!](\sigma) = \mathbf{T}$.

Definition 3–8: ($\models F$, validity of correctness formula F)
Given $F \in Form$, if $[\![F]\!](\sigma) = \mathbf{T}$ for all σ then F is said to be *valid*, written $\models F$.

This means that F is true with respect to our underlying structure. This does not say anything about the truth of F with respect to other interpretations of the function symbols, predicates, etc.

We can now formalize the idea of a *weakest precondition* of a statement and postcondition which we alluded to in Chapter 1, as well as the *strongest postcondition.*

Definition 3–9: ($wp(K,Q)$, weakest precondition)
An assertion $wp(K,Q)$ is said to be the *weakest precondition* for K resulting in postcondition Q if
(i) $\models \{wp(K,Q)\}K\{Q\}$
(ii) $\models \{P\}K\{Q\} \Rightarrow \models \{P \supset wp(K,Q)\}$

Definition 3–10: ($sp(K,P)$, strongest postcondition)
An assertion $sp(K,P)$ is said to be the *strongest postcondition* for K resulting from precondition P if

(i) $\models \{P\}K\{sp(K,P)\}$
(ii) $\models \{P\}K\{Q\} \Rightarrow \models \{sp(K,P) \supset Q\}$

The weakest precondition expresses exactly what has to be true in the state before execution of a code segment for the desired postcondition to hold. The strongest postcondition is the dual of this.

In Chapter 1, we said that we assume the *expressiveness* of our assertion language. By this, we mean that we assume that for any code segment K and postcondition Q, $wp(K,Q)$ can be written as an assertion in $Assn$, and similarly for any code segment K and precondition P, $sp(K,P)$ can be written as an assertion in $Assn$.

3.2. Inadequacy of Simple Assignment Axioms

Before presenting the assignment axiom, we show four examples of assignments. In each example, we determine what sort of assignment axiom would produce the proper precondition for a particular postcondition of an assignment statement. First we present a very simple example which is satisfied with the standard assignment axiom [21]. We then present Pascal type and variable declarations and an h–graph initial state which provide the setting for the successive and increasingly more difficult examples.

The simplest rule we present, the standard assignment axiom, is shown in example 3–2 to exhibit what we call the *component problem*. This rule does not work in the case that a pointer references a structure and we wish to make an argument about the value of a component of the structure after the pointer is assigned a value. To overcome this problem, we present a second, more complex rule. This second rule is shown to have the *alias problem* in that although it does not display the component problem when no aliases are present, it may fail if aliases are in effect. This is demonstrated in example 3–3. This leads us to a third, even more complex rule. However, the rule we develop to handle the simple alias case is shown in example 3–4 to fail to give the weakest precondition of an assignment when circular lists may be present. We finally develop a correct assignment

axiom, presented in Section 3.3, and show that it treats circular lists correctly. Section 3.4 gives the proof that the full axiom handles all cases correctly.

Example 3–1:

The first example involves two simple variables and an assignment. We assume the following Pascal declarations:

```
var x, y : integer;
```

This corresponds to the following h–graph initial state:

```
local–state: { g1:n1:[ # ]
                        – x –> n2:[ ? ]
                        – y –> n3:[ ? ]
             }
```

The structure of this state is important to us but the particular values of nodes n_2 and n_3 do not matter. We have left them unspecified here.

Consider the following Pascal program fragment:

```
x := 2;
y := x;
```

and the corresponding Hg program fragment:

```
/x := 2;
/y := /x;
```

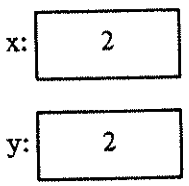


Figure 3.1 Program State after execution of Example 3–1.

local–state: { g_1:n_1:[#]
- x -> n_2:[2]
- y -> n_3:[2]
}

Figure 3.2 H–graph representation of Figure 3.1.

The result of executing the program fragments above would leave a Pascal state which can be represented as in Figure 3.1, and an h–graph which could be represented as in Figure 3.2. Suppose we want to determine a precondition for the following postcondition:

$/y = 2$

How can we generate a condition to insure this will be true after execution of

$/y := /x$

The assignment axiom used in most of the literature, which was introduced by Hoare [21], is the following:

$\{P_s^e\}\ s := e\ \{P\}$

The expression P_s^e represents the assertion which is identical to P except that every occurrence of s has been replaced by e.

In our example, performing this substitution gives us the precondition

$/x = 2$

which of course is true immediately before the assignment, so we would expect that the postcondition holds after the assignment, as it does in this simple case. This expectation would only be valid, however, if the assignment axiom we have chosen is a valid axiom for all assignments permitted by the language. Unfortunately this assignment axiom is inadequate, as the next example demonstrates.

Example 3–2:

The Pascal definitions and declarations we assume are the following:

```
type elementptr = ↑ element;
     element = record
                  head: integer;
                  tail: elementptr;
               end;
var p, q: elementptr;
```

Consider the following Pascal program fragment:

```
new(p);
p↑.head := 2;
p↑.tail := nil;
q := p;
```

and the corresponding Hg fragment:

```
newelement(/p);
/p/head := 2;
/p/tail := nil;
/q := /p;
```

The result of executing these fragments would leave a Pascal state as in Figure 3.3, and an h–graph as in Figure 3.4. Suppose we were interested in a precondition to the last assignment, $/q := /p$, which would guarantee that the assertion

$/q/head = 2$

holds in the final state. If we attempt to use the simple axiom shown above, we see that it is inadequate in this situation. Since there is no occurrence of the selector $/q$ in the assertion the substitution gives us the following precondition:

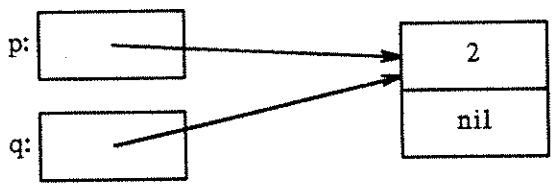


Figure 3.3 Program State after execution of Example 3–2.

local–state: { g_1:n_1:[#]
- p -> n_2:[g_2]
- q -> n_3:[g_2]

g_2: n_4:[#]
- *head* -> n_5:[2]
- *tail* -> n_6:[nil]

}

Figure 3.4 H–graph representation of Figure 3.3.

$$(/q/head=2)\Big|_q^p \equiv /q/head=2$$

which is false in the state preceding the assignment.

One might complain that indeed the selector $/q$ does appear in the postcondition as a prefix of the selector $/q/head$ and it should therefore be replaced by $/p$ in that context, resulting in the precondition $/p/head=2$, which would satisfy our intuition about what the precondition should be. The problem is slightly more complicated, however, since if we were to have a postcondition such as

$$/q.a/z=3$$

we would not want the $/q$ appearing in this assertion to be replaced with $/p$ to give us the precondition $/p.a/z=3$. Similarly we would not want postcondition

$$/x/q/z=3$$

to result in precondition $/x/p/z=3$. Clearly we would like an assignment axiom that would substitute $/p$ for $/q$ in exactly those cases where $/q$ is a prefix of a selector in the assertion. We can define such a substitution rule with little trouble. The substitution rule which follows is also employed in Chapter 4 in the procedure call rule.

First we formalize the notion of a selector prefix.

Definition 3–11: (prefix of a selector)
A selector s is said to be a *prefix* of another selector u written $prefix(s,u)$, if s is of the form $s_1 \cdots s_n$ each $s_i \in Gsel$, and u is of the form $s_1 \cdots s_n u_1 \cdots u_m$, where

$m \geqslant 0$ and each $u_i \in Gsel$. If $m > 0$ then s is said to be a *proper prefix* of u.

Definition 3–12: (substitution rule ∇)

We use q and u to represent selectors; x is a logical variable; d_1, and d_2 are terms; c is a constant; and P and Q are assertions. We assume that a selector u appearing in any alternative below is composed of graph selectors $u_1 \cdots u_m$ and that the selector s is composed of graph selectors $s_1 \cdots s_n$.

$\nabla_s^r(P)$ for any selector s, expression r, and assertion P is an assertion in *Assn* defined by the following cases of P:

The number labeling a case indicates which alternative of the definition for *Assn* is being defined (see Definition 3–2). The subcases of (7) are the alternatives of *Term* given by Definition 3–1.

(1): $\nabla_s^r(\exists x(P)) \equiv \exists x \nabla_s^r(P)$

(2): $\nabla_s^r(\neg P) \equiv \neg \nabla_s^r(P)$

(3): $\nabla_s^r(\rho(d_1, \cdots, d_n) \equiv \rho(\nabla_s^r(d_1), \cdots, \nabla_s^r(d_n))$

(4): $\nabla_s^r(P \wedge Q) \equiv \nabla_s^r(P) \wedge \nabla_s^r(Q)$

(5): $\nabla_s^r(P \vee Q) \equiv \nabla_s^r(P) \vee \nabla_s^r(Q)$

(6): $\nabla_s^r(P \supset Q) \equiv \nabla_s^r(P) \supset \nabla_s^r(Q)$

(7): $\nabla_s^r(d_1 = d_2) \equiv \nabla_s^r(d_1) = \nabla_s^r(d_2)$

(7.1): $$\nabla_s^r(u) \equiv \begin{cases} ru_{n+1} \cdots u_m & \text{if } r \in Sel \text{ and } u_1 \cdots u_n \equiv s,\ n < m \\ \perp & \text{if } r \notin Sel \text{ and } u_1 \cdots u_n \equiv s,\ n < m \\ r & \text{if } u \equiv s \\ u & \text{otherwise} \end{cases}$$

(7.2): $\nabla_s^r(x) \equiv x$

(7.3): $\nabla_s^r(c) \equiv c$

(7.4): $\nabla_s^r(f(d_1, \cdots, d_n)) \equiv f(\nabla_s^r(d_1), \cdots, \nabla_s^r(d_n))$

(8): $\nabla_s^r(eq(q,u)) \equiv \exists x(\nabla_s^r(eq(q,x)) \wedge \nabla_s^r(eq(u,x)))$

$$(9): \quad \nabla_s^r(eq(u,x)) \equiv \begin{cases} eq(ru_{i+1}\cdots u_m, x) & \text{if } u_1\cdots u_i \equiv s, \text{ for some } i<m \text{ and } r \in Sel \\ eq(\perp, x) & \text{if } u_1\cdots u_i \equiv s, \text{ for some } i<m \text{ and } r \notin Sel \\ eq(u,x) & \text{otherwise} \end{cases}$$

$$(10): \quad \nabla_s^r(eq(\perp,x)) \equiv eq(\perp,x)$$

Applying this substitution rule to postcondition $/q/head=2$ with assignment $/q=/p$, we see that the precondition of the assignment should be

$$/p/head=2 \equiv \nabla_{/q}^{/p}(/q/head=2)$$

Once again, although the new substitution rule is more complex and satisfies stronger conditions than the previous rule, it is important to determine if it holds true in all cases.

Example 3–3:

Examination of either of Figures 3.3 or 3.4 shows that the last example introduced an alias into the final state. Selectors /p/head and /q/head access the same node, as do /p/tail and /q/tail. Consider what would occur if the following Pascal assignment were made in that final state:

p↑.head := 3;

The corresponding Hg assignment is

$/p/head := 3;$

leading to the states shown in Figures 3.5 and 3.6.

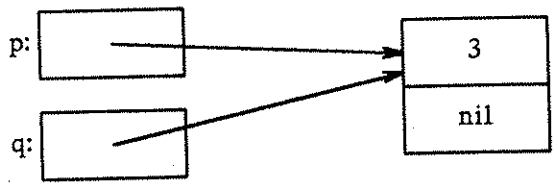


Figure 3.5 Program State after execution of Example 3–3.

local-state: { g_1:n_1:[#]
- p -> n_2:[g_2]
- q -> n_3:[g_2]

g_2: n_4:[#]
- *head* -> n_5:[3]
- *tail* -> n_6:[nil]

}

Figure 3.6 H-graph representation of Figure 3.5.

Now consider the same postcondition as in the last example, that is:

$/q/head = 2$

Our proposed assignment rule provides the precondition

$$\nabla^3_{/p/head}(/q/head = 2) \equiv /q/head = 2$$

This precondition is clearly not adequate since $/q/head$ does have the value 2 in the starting state (shown in Figure 3.4), but $/q/head = 2$ is not satisfied in the final state (shown in Figure 3.6).

The difficulty in this case occurs because although $/p$ is not textually identical to any prefix of $/q/head$, $/p$ selects the same node as $/q$ and therefore we must assure that any assertion about a selector with $/q$ as a proper prefix must be satisfied by the same selector with the value to be assigned $/p$ substituted for all occurrences of $/q$. That is, if $/p$ and $/q$ select the same node, then our axiom must provide a different precondition for an assertion involving a selector with proper prefix $/q$ than the assertion constructed by $\nabla^3_{/q}$.

Our new substitution rule, $\overline{\nabla}$ is quite similar to the rule ∇. The only modifications we need to make are changes to cases 3, 7, 9, and the subcases of 7. The formulae generated by $\overline{\nabla}$ are much more complex than for ∇. An application of $\overline{\nabla}$ to a formula takes each selector separately and generates all the possible cases of aliases between its prefixes and the selector on the left of the assignment. Each aliasing case leads to a different form

of term in the resulting assertion. As the length of the selectors in the postcondition determines the number of *eq* terms (alias tests) in the precondition, long selectors in the postcondition can lead to exponential growth in the size and complexity of the precondition.

Two technical devices are employed in the definition of $\overline{\nabla}$. Existentially quantified variables are introduced to allow selectors to be separated from more complex terms before expansion into the aliasing cases (as in case 7 below). Also, terms of the form $\uparrow(r,s)$ are introduced temporarily during the recursive substitution process, being replaced in the final result by either a selector or $\perp$. The same devices are used more extensively in the final assignment axiom in the next section.

Definition 3–13: (Substitution rule $\overline{\nabla}$,)
$\overline{\nabla}_s^r$ is defined for selectors s and expression r identically to ∇_s^r with the modifications to the cases shown below. We assume that $q \equiv q_1 \cdots q_m$.

(3): $\overline{\nabla}_s^r(\rho(d_1,\cdots,d_n)) \equiv \exists x_1,\cdots,x_n(\bigwedge_{1\leqslant i\leqslant n}(\overline{\nabla}_s^r(d_i=x_i) \wedge \rho(x_1,\cdots,x_n))$

(7): $\overline{\nabla}_s^r(d_1=d_2) \equiv \exists x(\overline{\nabla}_s^r(d_1=x) \wedge \overline{\nabla}_s^r(d_2=x))$

(7.1): $\overline{\nabla}_s^r(q=x) \equiv \bigvee_{1\leqslant i<m}(\bigwedge_{1\leqslant j<i}\neg eq(q_1\cdots q_j, s) \wedge eq(q_1\cdots q_i,s)$

$\wedge \overline{\nabla}_s^r(\uparrow(r,q_{i+1}\cdots q_m))=x)$

$\vee \bigwedge_{1\leqslant i<m}(\neg eq(q_1\cdots q_i, s) \wedge eq(q,s) \wedge r=x$

$\vee \bigwedge_{1\leqslant i\leqslant m}(\neg eq(q_1\cdots q_i, s) \wedge q=x$

(7.2): $\overline{\nabla}_s^r(x_1=x_2) \equiv x_1=x_2$

(7.3): $\overline{\nabla}_s^r(c=x) \equiv c=x$

(7.4): $\overline{\nabla}_s^r(f(d_1,\cdots,d_n)=x) \equiv \exists z_1,\cdots,z_n(\bigwedge_{1\leqslant i\leqslant n}\overline{\nabla}_s^r(d_i=z_i) \wedge f(z_1,\cdots z_n)=x)$

$$(9): \quad \overline{\nabla}_s^r(eq(q,x)) \equiv \bigvee_{1\leq i<m} \Big(\bigwedge_{1\leq j<i} \neg eq(q_1 \cdots q_j, s) \wedge eq(q_1 \cdots q_i, s) \wedge eq(\overline{\nabla}_s^r(\uparrow(r, q_{i+1} \cdots q_m)), x) \Big)$$

$$\vee \bigwedge_{1\leq i<m} (\neg eq(q_1 \cdots q_i, s) \wedge eq(q, x)$$

And we add the case:

$$(11): \quad \overline{\nabla}_s^r(\uparrow(r,q)) \equiv \begin{cases} rq & \text{if } r \in Sel \\ \perp & \text{if } r \notin Sel \end{cases}$$

We now apply $\overline{\nabla}$ to the postcondition, $/q/head = 2$, to see what precondition must be satisfied for the assignment $/p/head := 3$.

$$\overline{\nabla}^3_{/p/head}(/q/head = 2) \equiv \exists x (\overline{\nabla}^3_{/p/head}(/q/head = x) \wedge \overline{\nabla}^3_{/p/head}(2 = x))$$

$$\equiv \exists x, (eq(/q, /p/head) \wedge \perp = x$$

$$\vee \neg eq(/q, /p/head) \wedge eq(/q/head, /p/head) \wedge 3 = x$$

$$\vee \neg eq(/q, /p/head) \wedge \neg eq(/q/head, /p/head) \wedge /q/head = x$$

$$\wedge 2 = x)$$

Which can be simplified to read

$$\overline{\nabla}^3_{/p/head}(/q/head = 2) \equiv eq(/q, /p/head) \wedge \perp = 2$$

$$\vee \neg eq(/q, /p/head) \wedge eq(/q/head, /p/head) \wedge 3 = 2$$

$$\vee \neg eq(/q, /p/head) \wedge \neg eq(/q/head, /p/head) \wedge /q/head = 2$$

This precondition considers three distinct cases:

(1) In this case $/q$ is an alias of $/p/head$. Hence we must be able to follow the selector $/head$ from the value to be assigned to $/p/head$, 3, and find the value 2, but any selector through the value 3 must have value $\perp$. Clearly this can never be satisfied, so in such a starting state, the postcondition will never be satisfied.

(2) In this case $/q/head$ is an alias of $/p/head$. This is exactly the case in the state shown

in Figure 3.6. In such a case 3=2 must hold for the postcondition to be satisfied after the assignment. This does not hold, so the postcondition cannot be true after execution of the assignment in a state in which this alias prevails.

(3) In the final case, there are no aliases of */p/head*, and */q/head*=2 must be satisfied before the assignment in order for the postcondition to hold. This is the only aliasing pattern which will satisfy our postcondition for the assignment.

A new assignment axiom based on the substitution rule $\overline{\nabla}$ is getting closer to satisfying our needs in all situations. But it is also fairly complex. One might ask if such an axiom is really necessary. The answer depends on the program in which the axiom is to be applied. Looking back at our examples, we can see how to restrict the class of h-graph states to be able to use one of the simpler versions of the assignment axiom.

If we can guarantee that a program will manipulate no h-graph selectors of length greater than one graph selector (i.e., will use only simple variables), we can apply the simplest of the substitution rules in our assignment axiom. If we can guarantee that a program, though it may manipulate h-graph selectors of greater length, will never have two nodes sharing the same graph value (i.e., may use structured variables but no aliases), the assignment axiom can be based on the substitution rule ∇. If both structured variables and aliases may appear in our program, then we know we will need an axiom based on a substitution at least as complex as $\overline{\nabla}$. Once again, though, we are forced to ask whether or not $\overline{\nabla}$ is satisfactory for all programs one might wish to verify.

Example 3-4:

Figures 3.7 and 3.8 show a program state which can easily be constructed by a Pascal or Hg program. Suppose in this state we make the Pascal assignment

p↑.tail := p

or equivalently

/p/tail := */p*

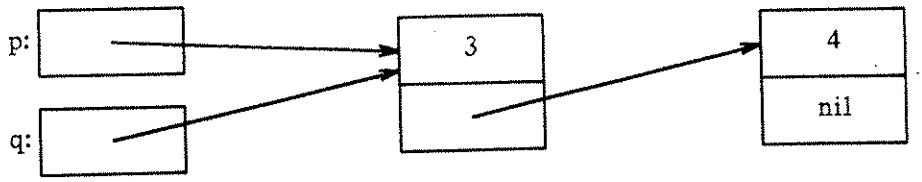


Figure 3.7 Program State **before** execution of Example 3–4.

local–state: { g_1: n_1:[#]
- p -> n_2:[g_2]
- q -> n_3:[g_2]

g_2: n_4:[#]
- *head* -> n_5:[3]
- *tail* -> n_6:[g_3]

g_3: n_7:[#]
- *head* -> n_8:[4]
- *tail* -> n_9:[nil]

}

Figure 3.8 H–graph representation of Figure 3.7.

in Hg. The result of execution of this statement is shown in Figures 3.9 and 3.10.

Suppose now that we want to develop a precondition for the assignment that will result in a state in which, similar to the state we have, the following postcondition holds:

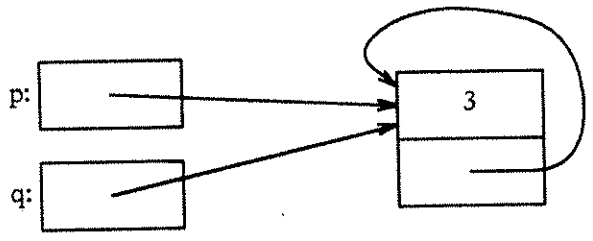


Figure 3.9 Program State **after** execution of Example 3–4.

local–state: { g_1:n_1:[#]
$- p \rightarrow n_2$:[g_2]
$- q \rightarrow n_3$:[g_2]

g_2:n_4:[#]
$- head \rightarrow n_5$:[3]
$- tail \rightarrow n_6$:[g_2]

}

Figure 3.10 H–graph representation of Figure 3.9.

$/q\,/tail\,/tail\,/head = 3$

We see that indeed, $\overline{\nabla}/^{p}_{q}$ does not produce a satisfactory precondition for the assignment.

$$\overline{\nabla}/^{p}_{p/tail}(/q\,/tail\,/tail\,/head = 3) \equiv \exists x(\overline{\nabla}/^{p}_{p/tail}(/q\,/tail\,/tail\,/head = x) \wedge \overline{\nabla}/^{p}_{p/tail}(3 = x)$$

$$\equiv \exists x \Big| eq(/q, /p\,/tail) \wedge /p\,/tail\,/tail\,/head = x$$

$$\vee \neg eq(/q, /p\,/tail) \wedge eq(/q\,/tail, /p\,/tail) \wedge /p\,/tail\,/head = x$$

$$\vee \neg eq(/q, /p\,/tail) \wedge \neg eq(/q\,/tail, /p\,/tail) \wedge eq(/q\,/tail\,/tail, /p\,/tail) \wedge /p/head = x$$

$$\vee \neg eq(/q, /p\,/tail) \wedge \neg eq(/q\,/tail, /p\,/tail) \wedge \neg eq(/q\,/tail\,/tail, /p\,/tail) \wedge$$
$$eq(/q\,/tail\,/tail\,/head, /p\,/tail) \wedge /p = x$$

$$\vee \neg eq(/q, /p\,/tail) \wedge \neg eq(/q\,/tail, /p\,/tail) \wedge \neg eq(/q\,/tail\,/tail, /p\,/tail) \wedge$$
$$\neg eq(/q\,/tail\,/tail\,/head, /p\,/tail) \wedge /q\,/tail\,/tail\,/head = x$$

$$\wedge (3 = x)$$

Let us consider the precondition only in the example state we have shown in Figures 3.7 and 3.8 as being the state before execution of the assignment. Since our starting state displays the condition $eq(/q\,/tail, /p\,/tail)$ the precondition reduces to

$$\neg eq(/q, /p\,/tail) \wedge eq(/q\,/tail, /p\,/tail) \wedge /p\,/tail\,/head = 3$$

which clearly does not hold in the state in which the assignment took place since

$/p/tail/head = 4$ in that state. The notable fact is that though the postcondition is true in in the resulting state, $\overline{\nabla}$ produces a precondition which is false in the starting state. Although it is true that were this precondition satisfied, the postcondition would be satisfied after the assignment, there are states where the precondition is not satisfied and the postcondition is satisfied, that is, we have not produced the *weakest precondition* for the given assignment statement. In this case, the complicating factor is that not only does $eq(/q/tail, /p/tail)$ hold in the state after execution of the assignment, but $eq(/q/tail/tail, /p/tail)$ holds as well and the precondition above does not address this fact.

3.3. The Assignment Axiom

We would now like to determine for any assignment statement $s := e$ and any assertion Q the weakest precondition P such that the correctness formula $\{P\}s := e\{Q\}$ is true, the one characterizing the largest set of states in which the correctness formula is true.

We considered several inadequate substitution rules. The substitution rule we introduce can be used to construct an axiom which correctly characterizes all assignment statements in Hg. Δ is quite similar to the substitution rule $\overline{\nabla}$ except that in cases where $\overline{\nabla}$ substitutes the right side selector for a prefix of a selector, Δ is applied recursively so that any circularities involving the left side selector are handled correctly. Oppen and Cook [33] employ a similar substitution rule in a language in which each program manipulates a single directed graph with nodes having atomic values. Since the recursive substitution of Δ is more complex than that found in $\overline{\nabla}$, the formulae generated by application of Δ are correspondingly more complex.

Definition 3–14: (substitution rule Δ)

We use q and u to represent selectors; x, y, and z are logical variables; d_1, and d_2 are terms; and P and Q are assertions. We assume that a selector u appearing

in any alternative below is composed of graph selectors $u_1 \cdots u_m$.

$\Delta_s^e(P)$ for any selector s, expression e, and assertion P is defined by the following cases of P:

The number labeling a case indicates which alternative of the definition for *Assn* is being defined. Cases (7.1) through (7.4) are for the different cases of the definition of *Term*. Cases (7.1.1) and (9.1) are auxiliary cases required for the definition of Δ_s^e.

(1): $\Delta_s^e(\exists x, (P)) \equiv \exists x, \Delta_s^e(P)$

(2): $\Delta_s^e(\neg P) \equiv \neg \Delta_s^e(P)$

(3): $\Delta_s^e(\rho(d_1, \cdots, d_n) \equiv \exists x_1, \cdots, x_n, \bigwedge_{1 \leqslant i \leqslant n} \Delta_s^e(d_i = x_i) \wedge \rho(x_1, \cdots, x_n)$

(4): $\Delta_s^e(P \wedge Q) \equiv \Delta_s^e(P) \wedge \Delta_s^e(Q)$

(5): $\Delta_s^e(P \vee Q) \equiv \Delta_s^e(P) \vee \Delta_s^e(Q)$

(6): $\Delta_s^e(P \supset Q) \equiv \Delta_s^e(P) \supset \Delta_s^e(Q)$

(7): $\Delta_s^e(d_1 = d_2) \equiv \exists x, \Delta_s^e(d_1 = x) \wedge \Delta_s^e(d_2 = x)$

(7.1): $$\Delta_s^e(u=x) \equiv \bigvee_{1 \leqslant i < m} \left| \bigwedge_{1 \leqslant j < i} (\neg eq(u_1 \cdots u_j, s)) \wedge eq(u_1 \cdots u_i, s) \right.$$
$$\left. \wedge \Delta_s^e(\uparrow(e, u_{i+1} \cdots u_m) = x) \right|$$
$$\vee \bigwedge_{1 \leqslant i < m} (\neg eq(u_1 \cdots u_i, s)) \wedge eq(u, s) \wedge e = x$$
$$\vee \bigwedge_{1 \leqslant i \leqslant m} (\neg eq(u_1 \cdots u_i, s)) \wedge u = x$$

(7.1.1): If $e \in Sel$, then

$$\Delta_s^e(\uparrow(e,u)=x) \equiv \bigvee_{1\leqslant i<m}\left[\bigwedge_{1\leqslant j<i}(\neg eq(eu_1\cdots u_j, s)) \wedge eq(eu_1\cdots u_i, s) \wedge \Delta_s^e(\uparrow(e,u_{i+1}\cdots u_m)=x)\right]$$

$$\vee \bigwedge_{1\leqslant i<m}(\neg eq(eu_1\cdots u_i, s)) \wedge eq(eu, s) \wedge e=x$$

$$\vee \bigwedge_{1\leqslant i\leqslant m}(\neg eq(eu_1\cdots u_i, s)) \wedge eu=x$$

If $e \notin Sel$ then

$$\Delta_s^e(\uparrow(e,u)=x) \equiv \mathbf{F}$$

(7.2): $\Delta_s^e(x_1 = x_2) \equiv x_1 = x_2$

(7.3): $\Delta_s^e(c = x) \equiv c = x$

(7.4): $\Delta_s^e(f(d_1,\cdots,d_n)=x) \equiv \exists x_1,\cdots,x_n, \bigwedge_{1\leqslant i\leqslant n}\Delta_s^e(d_i=x_i) \wedge f(x_1,\cdots,x_n)=x$

(8): $\Delta_s^e(eq(q,u)) \equiv \exists x, \Delta_s^e(eq(q,x)) \wedge \Delta_s^e(eq(u,x))$

(9): $\Delta_s^e(eq(u,x)) \equiv \bigvee_{1\leqslant i<m}\left[\bigwedge_{1\leqslant j<i}(\neg eq(u_1\cdots u_j,s)) \wedge eq(u_1\cdots u_i,s) \wedge \Delta_s^e(eq(\uparrow(e,u_{i+1}\cdots u_m),x))\right]$

$$\vee \bigwedge_{1\leqslant i<m}(\neg eq(u_1\cdots u_i,s)) \wedge eq(u,x)$$

(9.1): If $e \in Sel$, then

$$\Delta_s^e(eq(\uparrow(e,u),x)) \equiv \bigvee_{1 \leq i < m} \left| \bigwedge_{1 \leq j < i} (\neg eq(eu_1 \cdots u_j, s)) \wedge eq(eu_1 \cdots u_i, s) \right.$$
$$\left. \wedge \Delta_s^e(eq(\uparrow(e,u_{i+1} \cdots u_m),x)) \right|$$

$$\vee \bigwedge_{1 \leq i < m} (\neg eq(e\, u_1 \cdots u_i, s)) \wedge eq(eu, x)$$

If $e \notin Sel$ then

$$\Delta_s^e(eq(\uparrow(e,u),x)) \equiv \mathbf{F}$$

(10): $\nabla_s^r(eq(\perp,x)) \equiv eq(\perp,x)$

We define $\Delta_{\bar{s}}^{\bar{e}}$, the extension of Δ_s^e for vectors $\bar{s}$ and $\bar{e}$, in the same way that value substitution in a function is defined, that is left-to-right pairwise substitution. $\Delta_{\bar{s}}^{\bar{e}}(P)$ where $|\bar{s}|=|\bar{e}|=m$ is defined to be

$\Delta_{(\bar{s}\downarrow 2,\cdots,\bar{s}\downarrow m)}^{(\bar{e}\downarrow 2,\cdots,\bar{e}\downarrow m)}(\Delta_{\bar{s}\downarrow 1}^{\bar{e}\downarrow 1}(P))$ if $m>1$

$\Delta_{\bar{s}\downarrow 1}^{\bar{e}\downarrow 1}(P)$ if $m=1$

The assignment axiom provides the weakest precondition that makes a postcondition of an assignment statement true. The axiom is

$$\{\neg eq(s,\perp) \wedge \Delta_s^e(P)\} \quad s := e \quad \{P\}.$$

Note that we require that the selector on the left of the assignment be defined in any state satisfying the precondition. This avoids the possibility that the state $\perp$ might result from the assignment, causing the postcondition to be unsatisfied. Though we call this an axiom, it is actually an axiom scheme that can be applied to any given assignment by substituting the selector on the left hand side of the assignment for s and the expression for e.

We now apply the substitution Δ to Example 3–4, that is, the assignment $/p/tail := /p$ and the postcondition $/q/tail/tail/head = 3$. Recall that the precondition provided by $\overline{\nabla}$ was not the weakest precondition, and in particular is not satisfied in a state resulting in which the result of the assignment led to a circularity in $/q/tail$.

$$\Delta/^{p}_{p/tail}(/q/tail/tail/head = 3) = \exists x(\Delta/^{p}_{p/tail}(/q/tail/tail/head = x) \wedge \Delta/^{p}_{p/tail}(3 = x)$$

$$= \exists x \Big| \; eq(/q, /p/tail) \wedge \Delta/^{p}_{p/tail}(\uparrow(/p, /tail/tail/head = x)$$

$$\vee \neg eq(/q, /p/tail) \wedge eq(/q/tail, /p/tail) \wedge \Delta/^{p}_{p/tail}(\uparrow(/p, /tail/head) = x)$$

$$\vee \neg eq(/q, /p/tail) \wedge \neg eq(/q/tail, /p/tail) \wedge eq(/q/tail/tail, /p/tail) \wedge \Delta/^{p}_{p/tail}(\uparrow(/p, /head) = x)$$

$$\vee \neg eq(/q, /p/tail) \wedge \neg eq(/q/tail, /p/tail)$$

$$\wedge \neg eq(/q/tail/tail, /p/tail) \wedge eq(/q/tail/tail/head, /p/tail) \wedge /p = x$$

$$\vee \neg eq(/q, /p/tail) \wedge \neg eq(/q/tail, /p/tail) \wedge$$

$$\neg eq(/q/tail/tail, /p/tail) \wedge \neg eq(/q/tail/tail/head, /p/tail) \wedge /q/tail/tail/head = x \; \Big|$$

$$\wedge (3 = x)$$

Carrying through the applications of Δ and simplifying algebraically, one gets the following equation:

$$
\begin{aligned}
& eq(/q,/p/tail) \wedge \\
& \quad (eq(/p/head,/p/tail) \wedge /p=3 \\
& \quad \vee \neg eq(/p/head,/p/tail) \wedge /p/head=3) \\
\vee\ & \neg eq(/q,/p/tail) \wedge eq(/q/tail,/p/tail) \wedge \\
& \quad (eq(/p/head,/p/tail) \wedge /p=3 \\
& \quad \vee \neg eq(/p/head,/p/tail) \wedge /p/head=3) \\
\vee\ & \neg eq(/q,/p/tail) \wedge \neg eq(/q/tail,/p/tail) \wedge eq(/q/tail/tail,/p/tail) \wedge \\
& \quad (eq(/p/head,/p/tail) \wedge /p=3 \\
& \quad \vee \neg eq(/p/head,/p/tail) \wedge /p/head=3) \\
\vee\ & \neg eq(/q,/p/tail) \wedge \neg eq(/q/tail,/p/tail) \wedge \\
& \quad \neg eq(/q/tail/tail,/p/tail) \wedge eq(/q/tail/tail/head,/p/tail) \wedge /p=3 \\
\vee\ & \neg eq(/q,/p/tail) \wedge \neg eq(/q/tail,/p/tail) \wedge \\
& \quad \neg eq(/q/tail/tail,/p/tail) \wedge \neg eq(/q/tail/tail/head,/p/tail) \wedge /q/tail/tail/head=3
\end{aligned}
$$

This condition is satisfied in the starting state of Example 3–4, since in that state

$$\neg eq(/q,/p/tail) \wedge eq(/q/tail,/p/tail) \wedge \neg eq(/p/head,/p/tail) \wedge /p/head=3$$

holds. We show in Section 3.4 that, in fact, this assignment axiom provides the weakest precondition for any given postcondition of an assignment.

3.4. Proof of the Assignment Axiom

To simplify the proof of the assignment axiom we introduce several lemmas. The first states that if all prefixes of a selector u and the selector u itself are not aliased with another selector s in some state, then the value of u is the same if the value of the node selected by s is changed. The second states that if no prefix of a selector u is aliased with a selector s but u is aliased with s in some state, then substituting the value v for s in that state will change the value of u in that state to v as well.

Lemma 3-1: If u is a selector composed of graph selectors $u_1 \cdots u_m$ then for any state σ, selector s, and value $v \in \Omega \cup \Delta \hat{} \perp$,

$$[\![\neg eq(s,\perp) \wedge \bigwedge_{1 \leq i \leq m} \neg eq(u_1 \cdots u_i, s)]\!](\sigma) \Rightarrow R(u)(\sigma) = R(u)(\sigma\{v:s\}).$$

Proof: by induction on m.

Basis: Suppose $m=1$.

Then we must prove $[\![\neg eq(s,\perp) \wedge \neg eq(u_1,s)]\!](\sigma) \Rightarrow R(u)(\sigma) = R(u)(\sigma\{v:s\})$.

Let $\sigma = \langle G, V, r \rangle$.

Note that $[\![\neg eq(u_1,s)]\!](\sigma) \Rightarrow [\![u_1]\!](\sigma) \neq [\![s]\!](\sigma) \Rightarrow [\![u_1]\!](r) \neq [\![s]\!](\sigma)$.

We then readily see that

$R(u)(\sigma) = V([\![u]\!](\sigma))$

$= V\{v:[\![s]\!](\sigma)\}([\![u]\!](\sigma))$

which is valid since $u \equiv u_1$ and $[\![s]\!](\sigma) \neq [\![u_1]\!](\sigma)$ selects some node in state σ

$= V\{v:[\![s]\!](\sigma)\}([\![u]\!](r))$

$= V\{v:[\![s]\!](\sigma)\}([\![u]\!](\sigma\{v:s\}))$

definition of [[u]] and since $\sigma\{v:s\} = \langle G, V\{v:[\![s]\!](\sigma)\}, r \rangle$

$= R(u)(\sigma\{v:s\})$

Induction:

Suppose

$$[\![\neg eq(s,\perp) \wedge \bigwedge_{1 \leq i < m} \neg eq(u_1 \cdots u_i, s)]\!](\sigma) \Rightarrow R(u_1 \cdots u_{m-1})(\sigma) = R(u_1 \cdots u_{m-1})(\sigma\{v:s\})$$

and furthermore suppose $[\![\neg eq(u_1 \cdots u_m), s)]\!](\sigma)$.

Show that $[\![\neg eq(s,\perp) \wedge \bigwedge_{1 \leq i \leq m} \neg eq(u_1 \cdots u_i, s)]\!](\sigma) \Rightarrow R(u)(\sigma) = R(u)(\sigma\{v:s\})$.

$R(u)(\sigma) = V([\![u]\!](\sigma))$

$= V\{v:[\![s]\!](\sigma)\}([\![u]\!](\sigma))$

same reasoning as above

$= V\{v:[\![s]\!](\sigma)\}([\![u_m]\!](V([\![u_1 \cdots u_{m-1}]\!](\sigma))))$

$= V\{v:[\![s]\!](\sigma)\}([\![u_m]\!](R(u_1 \cdots u_{m-1})(\sigma)))$

definition of R

$= V\{v:[\![s]\!](\sigma)\}([\![u_m]\!](R(u_1 \cdots u_{m-1})(\sigma\{v:s\})))$

by induction

$= V\{v:[\![s]\!](\sigma)\}([\![u]\!](\sigma\{v:s\}))$

$= R(u)(\sigma\{v:s\})$

□

Lemma 3-2: If u is a selector composed of graph selectors $u_1 \cdots u_m$ then

$$[\![\neg eq(s,\perp) \wedge \bigwedge_{1 \leq i < m} (\neg eq(u_1 \cdots u_i, s)) \wedge eq(u,s)]\!](\sigma) \Rightarrow R(u)(\sigma\{v:s\}) = v.$$

Proof:

Suppose $[\![\neg eq(s,\perp) \wedge \bigwedge_{1 \leq i < m} (\neg eq(u_1 \cdots u_i,s)) \wedge eq(u,s)]\!](\sigma)$

$R(u)(\sigma\{v:s\}) = V\{v:[\![s]\!](\sigma)\}([\![u]\!](\sigma\{v:s\}))$
$= V\{v:[\![s]\!](\sigma)\}([\![u_m]\!](R(u_1 \cdots u_{m-1})(\sigma\{v:s\})))$
$= V\{v:[\![s]\!](\sigma)\}([\![u_m]\!](R(u_1 \cdots u_{m-1})(\sigma)))$
by our assumption and Lemma 3–1
$= V\{v:[\![s]\!](\sigma)\}([\![u]\!](\sigma))$
$= V\{v:[\![s]\!](\sigma)\}([\![s]\!](\sigma))$
since $[\![eq(u,s)]\!](\sigma)$
$= v$

□

The following lemma, which states a rather simple result about aliased selectors, can be proven by induction.

Lemma 3–3: If $[\![\bigwedge_{1 \leq j < i} \neg eq(u_1 \cdots u_j,s) \wedge eq(u_1 \cdots u_i,s)]\!](\sigma)$ then
$[\![u_1 \cdots u_i]\!](\sigma\{v:s\}) = [\![s]\!](\sigma)$

Proof: By induction.

□

Lemma 3–4: If $[\![\bigwedge_{1 \leq i < m} (\neg eq(u_1 \cdots u_i,s)]\!](\sigma)$ then
$[\![u_1 \cdots u_m]\!](V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))) = [\![eu_1 \cdots u_m]\!](\sigma)$

Proof: This can be easily verified by inductive application of Lemma 3–1.

□

The final preparatory lemma states an important fact about the terms of the Δ substitution involving the function ↑.

Lemma 3–5:
$[\![\Delta_s^e(\uparrow(e,u_1 \cdots u_m) = x]\!](\sigma) \Leftrightarrow R(u_1 \cdots u_m)(V\{R(e(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))) = x$

Proof: By induction on m.
Basis: Suppose m = 1. We refer to u_1 simply as u in this case.

$$[\![\, \Delta_s^e(\uparrow(e,u) = x)\,]\!](\sigma) \equiv [\![\, eq(eu, s) \wedge e = x \vee \neg eq(eu, s) \wedge eu = x\,]\!](\sigma)$$

It is clear that either $[\![\, eq(eu, s)\,]\!](\sigma)$ or $[\![\, \neg eq(eu, s)\,]\!](\sigma)$ will be true for any state σ. If we can show that both of

(1) $[\![\, eq(eu, s) \wedge e = x\,]\!](\sigma)$, and

(2) $[\![\, \neg eq(eu, s) \wedge eu = x\,]\!](\sigma)$

are true exactly when

$$R(u)(V\{R(e)(\sigma):[\![s [\![(\sigma)\}^+([\![s\,]\!](\sigma))) = x$$

then we are finished.

Proof of case (1): Suppose $[\![\, eq(eu, s)\,]\!](\sigma)$. Then

$$[\![eu\,]\!](\sigma) = [\![s\,]\!](\sigma)$$

and

$$R(u)(V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}^+([\![s\,]\!](\sigma)))$$

$$= V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![u\,]\!](V(R(e)(\sigma):[\![s\,]\!](\sigma)\}^+([\![s\,]\!](\sigma))))$$

$$= V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![u\,]\!](V(R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![s\,]\!](\sigma))))$$

by definition of V^+

$$= V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![u\,]\!](R(e)(\sigma)))$$

$$= V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![u\,]\!](V([\![e\,]\!](\sigma))))$$

$$= V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![eu\,]\!](\sigma))$$

$$= V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![s\,]\!](\sigma))$$

$$= R(e)(\sigma)$$

$$= x$$

Proof of case (2): Suppose $[\![\, \neg eq(eu, s)\,]\!](\sigma)$. Then

$$R(eu)(\sigma) = x$$

and

$$R(u)(V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}^+([\![s\,]\!](\sigma)))$$

$$= V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![u\,]\!](V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}^+([\![s\,]\!](\sigma)))$$

$$= V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![u\,]\!](V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![s\,]\!](\sigma)))$$

$$= V\{R(e)(\sigma):[\![s\,]\!](\sigma)\}([\![u\,]\!](R(e)(\sigma)))$$

$$= V\{R(e)(\sigma):[\![s]\!](\sigma)\}([\![u]\!](V([\![e]\!](\sigma))))$$

$$= V\{R(e)(\sigma):[\![s]\!](\sigma)\}([\![eu]\!](\sigma))$$

$$= V([\![eu]\!](\sigma))$$

since $[\![eu]\!](\sigma) \neq [\![s]\!](\sigma)$

$$= R(eu)(\sigma)$$

$$= x$$

Induction: Assume that for all i, $1 \leqslant i < m$, that the result

$[\![\Delta_s^e(\uparrow(e, u_1 \cdots u_i) = x]\!](\sigma) \Longleftrightarrow R(u_1 \cdots u_i)(V\{R(e(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))) = x$

holds. If we can prove that the result holds for $i=m$, then we are finished.

We know that exactly one of the following cases must hold:

(1) $\exists i, 1 \leqslant i < m, [\![\bigwedge_{1 \leqslant j < i} (\neg eq(eu_1 \cdots u_j, s)) \wedge eq(eu_1 \cdots u_i, s)]\!](\sigma)$, or

(2) $[\![\bigwedge_{1 \leqslant i < m} (\neg eq(eu_1 \cdots u_i, s)) \wedge eq(eu, s)]\!](\sigma)$, or

(3) $[\![\bigwedge_{1 \leqslant i \leqslant m} (\neg eq(eu_1 \cdots u_i, s))]\!](\sigma)$.

We consider each case separately and show that our result holds.

Proof of case (1):

$$R(u_1 \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma)))$$

$$= V\{R(e)(\sigma):[\![s]\!](\sigma)\}([\![u_1 \cdots u_m]\!](V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))))$$

$$= V\{R(e)(\sigma):[\![s]\!](\sigma)\}([\![u_{i+1} \cdots u_m]\!](V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+ ([\![u_1 \cdots u_i]\!](V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))))))$$

$$= V\{R(e)(\sigma):[\![s]\!](\sigma)\}([\![u_{i+1} \cdots u_m]\!](V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))))$$

by Lemma 3–3

$$= R(u_{i+1} \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma)))$$

By the inductive hypothesis,

$R(u_{i+1} \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))) = x \Longleftrightarrow [\![\Delta_s^e(\uparrow(e, u_{i+1} \cdots u_m)) = x]\!](\sigma)$

So in this case,

$$R(u_i \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))) = x$$

iff

$[\![\bigwedge_{1 \leqslant j < i} (\neg eq(eu_1 \cdots u_j, s)) \wedge eq(eu_1 \cdots u_i, s) \wedge \Delta_s^e(\uparrow(e, u_{i+1} \cdots u_m) = x)]\!](\sigma)$

Proof of case (2):

$$R(u_1 \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma)))$$

$$= V\{R(e)(\sigma):[\![s]\!](\sigma)\}([\![u_1 \cdots u_m]\!](V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))))$$

$$= V\{R(e)(\sigma):[\![s]\!](\sigma)\}([\![eu_1 \cdots u_m]\!](\sigma))$$

By Lemma 3–4

$$= V\{R(e)(\sigma):[\![s]\!](\sigma)\}([\![s]\!](\sigma))$$

$$= R(e)(\sigma)$$

So, if $R(u_1 \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma)))$ is to have the value x, we must have $R(e)(\sigma)=x$, and we now have

$$[\![\bigwedge_{1 \leqslant i < m} (\neg eq(eu_1 \cdots u_i, s)) \wedge eq(eu, s) \wedge e = x]\!](\sigma)$$

iff

$$R(u_1 \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))) = x$$

Proof of case (3): Suppose

$$[\![\bigwedge_{1 \leqslant i \leqslant m} (\neg eq(eu_1 \cdots u_i, s))]\!](\sigma)$$

Then we have

$$R(u_1 \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma)))$$

$$= V\{R(e)(\sigma):[\![s]\!](\sigma)\}([\![u_1 \cdots u_m]\!](V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))))$$

$$= V\{R(e)(\sigma):[\![s]\!](\sigma)\}([\![eu_1 \cdots u_m]\!](\sigma))$$

By Lemma 3–4

$$= R([\![eu_1 \cdots u_m]\!](\sigma))$$

And, as above

$$R(u_1 \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))) = x$$

iff

$$[\![\bigwedge_{1 \leqslant i \leqslant m} (\neg eq(eu_1 \cdots u_i, s)) \wedge eq = x]\!](\sigma)$$

Since we have exhausted all possibilites, we know that

$$[\![\Delta_s^e(\uparrow(e, u_1 \cdots u_m) = x)]\!](\sigma) \iff R(u_1 \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))) = x$$

□

Theorem 3–6: Correctness of assignment statements

$$\{\neg eq(s,\perp) \wedge \Delta_s^e(P)\}s := e\{P\}$$

Proof: To prove this, we must show that $[\![\neg eq(s,\perp) \wedge \Delta_s^e(P)]\!](\sigma) \Rightarrow [\![P]\!](\sigma\{R(e)(\sigma):s\})$ for arbitrary σ. We will however prove a stronger result, that $[\![\neg eq(s,\perp) \wedge \Delta_s^e(P)]\!](\sigma) \Leftrightarrow [\![P]\!](\sigma\{R(e)(\sigma):s\})$ for arbitrary σ satisfying $\sigma\{R(e)(\sigma):s\} \neq \perp$. This stronger result demonstrates the desired property that Δ_s^e is not only a valid precondition for the assignment, but it is the weakest valid precondition.

The first several cases of Δ_s^e are uninteresting and straightforward. It is the cases of $u=x$ and $eq(u,x)$ that we are concerned with showing how to prove here. We will show how to prove the case for $u=x$. The other case can be proved analogously.

Hence we must show that

$$[\![\neg eq(s,\perp) \wedge \bigvee_{1 \leqslant i < m} \left(\bigwedge_{1 \leqslant j < i} (\neg eq(u_1 \cdots u_j, s)) \wedge eq(u_1 \cdots u_i, s) \right.$$

$$\left. \wedge \Delta_s^e(\uparrow(e, u_{i+1} \cdots u_m) = x) \right)$$

$$\vee \bigwedge_{1 \leqslant i < m} (\neg eq(u_1 \cdots u_i, s)) \wedge eq(u,s) \wedge e = x$$

$$\vee \bigwedge_{1 \leqslant i \leqslant m} (\neg eq(u_1 \cdots u_i, s)) \wedge u = x]\!](\sigma)$$

$$\Leftrightarrow [\![u = x]\!](\sigma\{R(e)(\sigma):s\})$$

If: The proof proceeds by simultaneous induction on the cases of P in the definition of Δ_s^e.

The left hand side consists of a number of disjunctions of conjunctions. These disjunctions can be broken up into three groups. In the first group, for some proper prefix u' of the selector u, none of the prefixes of u' is aliased with s, but u' is aliased with s. The second group consists of the case in which u itself is aliased with s but none of the prefixes of u is an alias. The last disjunction requires that none of the prefixes of u are aliases of s and u itself is not an alias of s. Clearly these are disjoint conditions, at most one of them may be true. Hence we can divide the proof into three cases:

(1) some prefix of u but not u itself is an alias of s in state σ

(2) u itself is an alias of s in σ.

(3) all the prefixes of u and u itself are not aliases of s in σ.

Proof of case (1):

In this case we must show that if for some $i < m$

$$[\![\neg eq(s,\perp) \wedge \bigwedge_{1 \leqslant j < i} (\neg eq(u_1 \cdots u_j, s)) \wedge eq(u_1 \cdots u_i, s) \wedge \Delta_s^e(\uparrow(e, u_{i+1} \cdots u_m) = x)]\!](\sigma) \quad (*)$$

then

$$[\![u = x]\!](\sigma\{R(e)(\sigma):s\})$$

Note that since

$$(*) \Rightarrow [\![\neg eq(s,\perp) \wedge \Delta_s^e(\uparrow(e, u_{i+1} \cdots u_m) = x)]\!](\sigma)$$

$$\Rightarrow R(u_{i+1} \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma))) = x$$

By Lemma 3–5.

$$\Rightarrow R(u_{i+1} \cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![u_1 \cdots u_i]\!](\sigma\{R(e)(\sigma):s\}))) = x$$

By Lemma 3–3.

$$\Rightarrow R(u_1 \cdots u_m)(\sigma\{R(e)(\sigma):s\}) = x$$

By Lemma 2–2.

$$\Rightarrow [\![u = x]\!](\sigma\{R(e)(\sigma):s\})$$

Proof of case (2):

In this case we must show that

$$[\![\neg eq(s,\perp) \wedge \bigwedge_{1 \leqslant i < m} (\neg eq(u_1 \cdots u_i, s)) \wedge eq(u,s) \wedge e = x]\!](\sigma)$$

implies

$$[\![u = x]\!](\sigma\{R(e)(\sigma):s\})$$

We can apply Lemma 3–2 to get

$$R(u)(\sigma\{R(e)(\sigma):s\}) = R(e)(\sigma)$$

And since $[\![e = x]\!](\sigma) \Rightarrow R(e)(\sigma) = x$, we have

$$R(u)(\sigma\{R(e)(\sigma):s\}) = x$$

$$\Rightarrow [\![u = x]\!](\sigma\{R(e)(\sigma):s\})$$

Proof of case (3):

Since we have

$$[\![\neg eq(s,\perp) \wedge \bigwedge_{1 \leqslant i \leqslant m} (\neg eq(u_1 \cdots u_i, s)) \wedge u = x]\!](\sigma)$$

we know by Lemma 3–1 that

$$R(u)(\sigma)=R(u)(\sigma\{R(e)(\sigma):s\})$$

and since the value of the logical variable x is independent of σ, we know that

$$[\![x]\!](\sigma)=[\![x]\!](\sigma\{R(e)(\sigma):s\})$$

we then have

$$[\![u=x]\!](\sigma\{R(e)(\sigma):s\})$$

Only If: Once again we prove this by simultaneous induction, and again we have three cases:

(1) some prefix of u but not u itself is an alias of s in state σ,

(2) u itself is an alias of s in σ,

(3) all the prefixes of u and u itself are not aliases of s in σ.

Proof of case (1):

In this case, we must show that if

$$[\![u=x]\!](\sigma\{R(e)(\sigma):s\})$$

and for some $i<m$,

$$[\![\bigwedge_{1\leqslant j<i}(\neg eq(u_1\cdots u_j,s))\wedge eq(u_1\cdots u_i,s)]\!](\sigma),$$

that

$$[\![\neg eq(s,\perp)\wedge \bigwedge_{1\leqslant j<i}(\neg eq(u_1\cdots u_j,s))\wedge eq(u_1\cdots u_i,s)\wedge \Delta_s^e(\uparrow(e,u_{i+1}\cdots u_m)=x)]\!](\sigma)$$

Given that $\sigma\{R(e)(\sigma):s\}\neq\perp$ and the two stated assumptions, Lemma 3–2 shows that

$$[\![\neg eq(s,\perp)]\!](\sigma)\wedge R(u_{i+1}\cdots u_m)(V\{R(e)(\sigma):[\![s]\!](\sigma)\}^+([\![s]\!](\sigma)))=x$$

which, by Lemma 3–5, clearly implies

$$[\![\neg eq(s,\perp)]\!](\sigma)\wedge[\![\Delta_s^e(\uparrow(e,u_{i+1}\cdots u_m)=x]\!](\sigma)$$

So we have

$$[\![\bigwedge_{1\leqslant j<i}(\neg eq(u_1\cdots u_j,s))\wedge eq(u_1\cdots u_i,s)]\!](\sigma)$$
$$\wedge [\![\neg eq(s,\perp)]\!](\sigma)\wedge[\![\Delta_s^e(\uparrow(e,u_{i+1}\cdots u_m)=x]\!](\sigma)$$

which is equal to

$$[\![\neg eq(s,\perp)\wedge \bigwedge_{1\leqslant j<i}(\neg eq(u_1\cdots u_j,s))\wedge eq(u_1\cdots u_i,s)\wedge \Delta_s^e(\uparrow(e,u_{i+1}\cdots u_m)=x)]\!](\sigma)$$

Proof of case (2):

We must show that if

$$[\![u=x]\!](\sigma\{R(e)(\sigma):s\})$$

and

$$[\![\bigwedge_{1 \le i < m} (\neg eq(u_1 \cdots u_i, s)) \wedge eq(u, s)]\!](\sigma)$$

then

$$[\![\neg eq(s, \perp) \wedge \bigwedge_{1 \le i < m} (\neg eq(u_1 \cdots u_i, s)) \wedge eq(u, s) \wedge e = x]\!](\sigma)$$

Given that $\sigma\{R(e)(\sigma):s\} \neq \perp$ and our assumption above,

$$[\![\neg eq(s, \perp) \wedge \bigwedge_{1 \le i < m} (\neg eq(u_1 \cdots u_i, s)) \wedge eq(u, s)]\!](\sigma)$$

Which implies by Lemma 3–2:

$$R(u)(\sigma\{R(e)(\sigma):s\}) = R(e)(\sigma) \qquad (\dagger)$$

And since

$$[\![u = x]\!](\sigma\{R(e)(\sigma):s\}) \equiv R(u)(\sigma\{R(e)(\sigma):s\}) = x$$

Substituting in (†) above we get

$$R(e)(\sigma) = x$$

or

$$[\![e = x]\!](\sigma)$$

which together with our assumptions gives us

$$[\![\neg eq(s, \perp) \wedge \bigwedge_{1 \le i < m} (\neg eq(u_1 \cdots u_i, s)) \wedge eq(u, s) \wedge e = x]\!](\sigma)$$

Proof of case (3): identical to case (3) above.

□

Corollary 3–7:
If $\models \{P\} s := e \{Q\}$ then $\models \{P \supset \neg eq(s, \perp) \wedge \Delta_s^e(Q)\}$

Proof: By our premise,

$$\forall \sigma \in Stat, [\![P]\!](\sigma) \supset [\![Q]\!](Last(Comp(s := e)(\sigma)))$$
$$= [\![Q]\!](\sigma\{R(e)(\sigma):s\})$$

And by Theorem 3–6

$$[\![Q]\!](\sigma\{R(e)(\sigma):s\}) \supset [\![\neg eq(s, \perp) \wedge \Delta_s^e(Q)]\!](\sigma)$$

so we have that for all states σ, $[\![P]\!](\sigma) \supset [\![\neg eq(s, \perp) \wedge \Delta_s^e(Q)]\!](\sigma)$ or $\models \{P \supset \neg eq(s, \perp) \wedge \Delta_s^e(Q)\}$

□

Having proved the correctness of the assignment rule permits us to prove the theorem which follows. This theorem and a similar one for procedure calls will serve as a basis for simplifying proofs of programs involving aliases.

Theorem 3–8: If $\sigma=\langle G,V,r\rangle$ and $[\![s]\!](\sigma) = n$, $n \in nodeset(r)$, and

$$\forall u_1 \in Sel\sim\{s\}, [\![\neg eq(u_1,s)]\!](\sigma)$$

and

$$\sigma' = Last(Comp(u_2 := e)(\sigma))$$

then

$$\forall u_1 \in Sel\sim\{s\}, [\![\neg eq(u_1,s)]\!](\sigma')$$

In other words, if the selector s selects a node in the root graph of state σ, and s has no aliases in σ, then assigning any expression's value to any node in σ cannot cause an alias of s to exist.

Proof: by contradiction

Suppose

$$[\![eq(u_1,s)]\!](\sigma') = [\![eq(u_1,s)]\!](\sigma\{R(e)(\sigma):u_2\})$$

Then we know from the assignment rule that

$$[\![\Delta^e_{u_2}(eq(u_1,s))]\!](\sigma)$$

Which, following expansion of Δ, is equivalent to

$$\exists x, [\![\Delta^e_{u_2}(eq(u_1,x))]\!](\sigma) \wedge [\![\Delta^e_{u_2}(eq(s,x))]\!](\sigma)$$

Since $[\![s]\!](\sigma)$ is in the rootgraph of σ, and is unaliased, it is easy to show that

$$[\![\Delta^e_{u_2}(eq(s,x))]\!](\sigma)$$

is satisfied only if

$$[\![eq(s,x)]\!](\sigma)$$

Expanding the expression

$$[\![\Delta^e_{u_2}(eq(u_1,x))]\!](\sigma)$$

we get, assuming $u_1 \equiv gsel_1 \cdots gsel_m$:

$$[\![\bigvee_{1\leq i<m} \left(\bigwedge_{1\leq j<i} (\neg eq(gsel_1 \cdots gsel_j, u_2)) \wedge eq(gsel_1 \cdots gsel_i, u_2) \right.$$

$$\left. \wedge \Delta^e_s(eq(\uparrow(e, gsel_{i+1} \cdots gsel_m), x)) \right)$$

$$\vee \bigwedge_{1\leq i<m} (\neg eq(gsel_1 \cdots gsel_i, u_2)) \wedge eq(u_1,x)]\!](\sigma)$$

Since we know that $[\![eq(u_1,x)]\!](\sigma)$ cannot be true when $[\![eq(s,x)]\!](\sigma)$, we can eliminate the last conjunction term in the above expression from consideration. We are now left with the task of showing that for any i, $1 \leq i < m$,

$$[\![\bigwedge_{1\leq j<i} (\neg eq(gsel_1 \cdots gsel_j, u_2)) \wedge eq(gsel_1 \cdots gsel_i, u_2)$$

$$\wedge \ \Delta_s^e(eq(\uparrow(e, gsel_{i+1} \cdots gsel_m), x))[\![\sigma)$$

Inspection of Δ_s^e reveals that for

$$[\![\Delta_s^e(eq(\uparrow(e, gsel_{i+1} \cdots gsel_m), x))]\!](\sigma)$$

to hold, e must be a selector u_4, and there must be some j, $i+1 \leqslant j \leqslant m$, such that

$$[\![eq(u_4 gsel_j \cdots gsel_m, x)]\!](\sigma)$$

So this selector $u_4 gsel_j \cdots gsel_m$ selects the same node as s in state σ, a contradiction, giving us the desired result.

□

3.5. Chapter Summary

In Section 1, we define the language of assertions on which we base the correctness formulae used to make arguments about Hg programs. We look at assignment axioms in Section 2. Though they grow increasingly more complex, none of these axioms successfully generates a precondition for every case of Hg assignments. We develop an assignment axiom in Section 3 which captures the behavior of assignments in even the most complex cases of aliasing and show in Section 4 that this assignment axiom is not only correct, but also provides the best possible result by describing the weakest preconditions of any assignment statement. We leave it to Chapter 4 to develop a proof system for Hg based on this assignment axiom.

Chapter 4

Verifying Hg Programs

In Chapter 3 we present an assignment axiom which lets us determine the weakest precondition for any Hg assignment statement and postcondition. We show this axiom is both sound and complete for Hg assignments. In this chapter we present a method by which we can build proofs of entire programs.

In the section which follows we introduce the reader to the idea of an inference rule and develop inference rules for all statements except procedure calls. In addition, we define what we mean by a formal proof and present a system of formal proof for the language Hg with no procedure calls. We show that this system is both *sound* and *relatively complete.* After that, we give an example of a program proof using this system. In the section following that we develop an inference rule which lets us prove properties of a certain class of procedure calls and extend our formal proof system to incorporate this proof rule, resulting in a system which is still sound and complete. We then show how properties of an actual procedure call are proven using this rule.

4.1. Inference Rules and the Language Hg

The proof system for Hg programs consists of the assignment axiom proved in the preceding chapter and a set of inference rules we provide here. An inference rule supports the construction of new valid formulae from known valid formulae. We provide an inference rule in this section for each of the statement types of the language aside from assignment and procedure call. Each alternative is presented in a lemma which demonstrates its soundness.

Definition 4–1: (inference and soundness)

(1) An *inference* is a construct of the form $\frac{F_1,\cdots F_n}{F}$ where each F_i, and $F \in Form$.

(2) An inference $\frac{F_1,\cdots F_n}{F}$ is said to be *sound* if $(\forall i, 1 \leqslant i \leqslant n, \models F_i) \Rightarrow \models F$

In other words $\frac{F_1,\cdots F_n}{F}$ is sound whenever if for all σ and for all $1 \leqslant i \leqslant n$, $[\![F_i]\!](\sigma)$, then for all σ, $[\![F]\!](\sigma)$.

Recall from Chapter 3 that

$$\models \{P\}K\{Q\} \text{ iff } \forall \sigma \in Stat, [\![P]\!](\sigma) \wedge |Comp(K)(\sigma)| \in \mathbf{N} \supset [\![Q]\!](Last(Comp(K)(\sigma)))$$

This captures the idea that a formula involving the code segment K is valid if it is true for all states in which the execution of K terminates. Bearing this in mind, we omit the term $|Comp(K)(\sigma)| \in \mathbf{N}$ in all of the proofs below, restricting ourselves to looking at only the cases where the code segment terminates.

The sequential composition rule lets us prove arguments about sequences of statements separated by semicolons in a program.

Lemma 4–1: (sequential composition)

$$\frac{\{P_1\}k\{P_2\},\{P_2\}K\{P_3\}}{\{P_1\}k;K\{P_3\}}$$

Proof: Recall that

$$\models \{P\}K\{Q\} \text{ iff } \forall \sigma \in Stat, [\![P]\!](\sigma) \Rightarrow [\![Q]\!](Last(Comp(K)(\sigma)))$$

Suppose that for all σ,

$$[\![P_1]\!](\sigma) \Rightarrow [\![P_2]\!](Last(Comp(k)(\sigma)))$$

and for all σ,

$$[\![P_2]\!](\sigma) \Rightarrow [\![P_3]\!](Last(Comp(K)(\sigma))).$$

Then in particular

$$[\![P_1]\!](\sigma) \Rightarrow [\![P_2]\!](Last(Comp(k)(\sigma))) \Rightarrow [\![P_3]\!](Last(Comp(K)(Last(Comp(k)(\sigma)))))$$
$$\equiv [\![P_3]\!](Last(Comp(k;K)(\sigma))).$$

So we have

$$\{P_1\}k;K\{P_3\}$$

□

The function $def : Bexp \rightarrow (Stat \rightarrow \{ \mathbf{T}, \mathbf{F} \})$ tells us whether a boolean expression is defined in a certain state or is undefined. Function def is defined simply to rule out the possibility that $B(b)(\sigma) = \mathbf{U}$.

Definition 4–2: (def : definite value function)

$$def\,(b) = (b \vee \neg b)$$

We require that $def\,(b)$ hold in the state before execution of any control structure with condition b if we want to make any argument about correctness. This is required because in the case where $\neg def\,(b)$, the execution sequence of the control structure is defined by $Comp$ to result in the single state $\perp$. Since no nontrivial assertions can hold in the state $\perp$, we must force the precondition P of an assertion about such a state to be false in order to guarantee that any formula $\{P\}K\{Q\}$ be true as well.

Before considering inference rules for control structures, we need to note that a Boolean expression has the same meaning whether it appears in a control statement or an assertion, that is,

$$\forall \sigma \in Stat,\ b \in Bool,\ B(b)(\sigma) = [\![b]\!](\sigma).$$

This is easily verified by consulting Definitions 2–21 and 3–4.

The following inference rule expresses what can be demonstrated about **if** statements. Notice the use of $def\,(b)$ in this rule.

Lemma 4–2: (conditional statements)

$$\frac{\{P \wedge b\}K_1\{Q\},\ \{P \wedge \neg b\}K_2\{Q\}}{\{P \wedge def\,(b)\}\ \textbf{if}\ b\ \textbf{then}\ K_1\ \textbf{else}\ K_2\ \textbf{endif}\ \{Q\}}$$

Proof: For any state σ,

$$[\![P \wedge b]\!](\sigma) \Rightarrow [\![Q]\!](Last(Comp(K_1)(\sigma)))$$

and

$$[\![P \wedge \neg b]\!](\sigma) \Rightarrow [\![Q]\!](Last(Comp(K_2)(\sigma))).$$

Suppose in some particular state σ, $[\![P \wedge def\,(b)]\!](\sigma)$.

$$[\![P \wedge def\,(b)]\!](\sigma) \Rightarrow [\![P \wedge b]\!](\sigma) \vee [\![P \wedge \neg b]\!](\sigma)$$

If $[\![b]\!](\sigma)$ is true then

$Last(Comp(\ \textbf{if}\ b\ \textbf{then}\ K_1\ \textbf{else}\ K_2\ \textbf{endif}\ )(\sigma)) \equiv Last(Comp(K_1)(\sigma))$,

and we know that

$$[\![P \wedge b]\!](\sigma) \Rightarrow [\![Q]\!](Last(Comp(K_1)(\sigma)))$$

hence

$$[\![P \wedge b]\!](\sigma) \Rightarrow [\![Q]\!](Last(Comp(\ \textbf{if}\ b\ \textbf{then}\ K_1\ \textbf{else}\ K_2\ \textbf{endif}\ )(\sigma))).$$

If $[\![\neg b]\!](\sigma)$ is true then

$Last(Comp(\ \textbf{if}\ b\ \textbf{then}\ K_1\ \textbf{else}\ K_2\ \textbf{endif}\ )(\sigma)) \equiv Last(Comp(K_2)(\sigma))$,

and we know that

$$[\![P \wedge \neg b]\!](\sigma) \Rightarrow [\![Q]\!](Last(Comp(K_2)(\sigma)))$$

so we know that

$$[\![P \wedge \neg b]\!](\sigma) \Rightarrow [\![Q]\!](Last(Comp(\ \textbf{if}\ b\ \textbf{then}\ K_1\ \textbf{else}\ K_2\ \textbf{endif}\ )(\sigma))).$$

Since we have exhausted the possibilities for the value of $[\![b]\!](\sigma)$, we have shown that

$$[\![P \wedge def(b)]\!](\sigma) \Rightarrow [\![Q]\!](Last(Comp(\ \textbf{if}\ b\ \textbf{then}\ K_1\ \textbf{else}\ K_2\ \textbf{endif}\ )(\sigma))),$$

or

$$\{P \wedge def(b)\}\ \textbf{if}\ b\ \textbf{then}\ K_1\ \textbf{else}\ K_2\ \textbf{endif}\ \{Q\}$$

□

The if rule above is the one which would normally be used in developing a program proof. One would first prove the subcomponent formulae, then apply the if rule to get a formula about the if statement. The following rule is of no practical use in program proofs, but it is introduced for use later in our demonstration of the completeness of the proof system developed here. The proof of it is similar to the proof of the if statement above.

Corollary 4–3:

$$\frac{\{P \wedge def(b)\}\ \textbf{if}\ b\ \textbf{then}\ K_1\ \textbf{else}\ K_2\ \textbf{endif}\ \{Q\}}{\{P \wedge b\}K_1\{Q\} \wedge \{P \wedge \neg b\}K_2\{Q\}}$$

Proof: Left to the reader.

□

The next inference rule, that of consequence, is perhaps the most important inference rule, since it permits us to make arguments about a program based on the truth of assertions in our underlying logic. If an assertion P implies the truth of P_1 in the underlying logic, then P is a valid precondition for any formula which has P_1 as a precondition. And if the truth of Q_1 always implies the truth of Q, then Q is a valid postcondition for any formula having Q_1 as a postcondition.

Lemma 4–4: (consequence)

$$\frac{\{P \supset P_1\},\{P_1\}K\{Q_1\},\{Q_1 \supset Q\}}{\{P\}K\{Q\}}$$

Proof: We assume that for all σ, $[\![P \supset P_1]\!](\sigma)$, $[\![\{P_1\}K\{Q_1\}]\!](\sigma)$, and $[\![Q_1 \supset Q]\!](\sigma)$. Note then that for arbitrary σ,

$$\begin{aligned}&[\![P]\!](\sigma) \Rightarrow [\![P_1]\!](\sigma)\\ &\Rightarrow [\![Q_1]\!](Last(Comp(K)(\sigma)))\\ &\Rightarrow [\![Q]\!](Last(Comp(K)(\sigma)))\end{aligned}$$

so by definition

$$\{P\}K\{Q\}$$

□

We now present the inference rule for while statements. Recall once again that a formula is considered valid if it holds for all states starting from which execution of the code segment will terminate. It is interesting to note the appearance of $def(b)$ in the premise as well. For the execution of a while statement to be defined, it must be the case that execution of the body cannot lead to a state in which the condition is undefined. If this were ever the case, the resulting state would be $\perp$, leading to an invalid correctness formula. Such a situation might arise, for instance, if one were to make an assignment which would cause a selector in the condition to select $\perp$.

Lemma 4–5: (while statements)

$$\frac{\{P \wedge b\}K\{P \wedge def(b)\}}{\{P \wedge def(b)\}\ \textbf{while}\ b\ \textbf{loop}\ K\ \textbf{endloop}\ \{P \wedge \neg b\}}$$

Proof: We assume that for all σ ,

$$[\![P \wedge b]\!](\sigma) \Rightarrow [\![P \wedge def\ (b)]\!](Last(Comp(K)(\sigma))).$$

We must show that for all σ, if $[\![P \wedge def\ (b)]\!](\sigma)$ then

$$[\![P \wedge \neg b]\!](Last(Comp(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(\sigma))).$$

The proof is by induction on the length of the sequence generated by *Comp*.

Basis:

Suppose $Comp(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(\sigma)$ is of length 1, then by inspection of *Comp*, $[\![P \wedge \neg b]\!](\sigma)$ must hold. Then

$$Last(Comp(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(\sigma)) = \sigma$$

by inspection of *Comp*. And clearly

$$[\![P \wedge \neg b]\!](Last(Comp(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(\sigma))).$$

Induction:

Suppose $Comp(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(\sigma)$ is of length greater than 1. Then

$$Comp(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(\sigma)$$
$$= \langle\sigma\rangle^{\frown} Comp(K)(\sigma)^{\frown} Comp(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(Last(Comp(K)(\sigma)))$$

Now we know by assumption that $[\![P \wedge def\ (b)]\!](Last(Comp(K)(\sigma)))$, and so by induction we have that

$$[\![P \wedge \neg b]\!](Last(Comp(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(Last(Comp(K)(\sigma))))$$
$$= [\![P \wedge \neg b]\!](Last(Comp(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(\sigma)))$$

Therefore we have for arbitrary σ

$$[\![P \wedge def\ (b)]\!](\sigma) \Rightarrow [\![P \wedge \neg b]\!](Last(Comp(\textbf{ while } b \textbf{ loop } K \textbf{ endloop })(\sigma)))$$

□

Now that we have soundness results for several inference rules, we give a more precise definition of the concept of a formal proof in this setting, so that we may employ these inference rules in a meaningful way. In Chapter 5 we extend this concept to include proofs that involve arguments about data types.

Definition 4–3: (formal proof)
Given a set of correctness formulae Ax, called the axioms, and a set of inference rules Pr, called the proof rules, we say that F is *formally provable* from Ax and Pr, written

$$\vdash_{Ax,Pr} F$$

whenever there exists $n \geqslant 1$ and a sequence of correctness formulae $F_1, \cdots F_n$ (called the formal proof of F) such that

(1) $F \equiv F_n$

(2) for each i, $1 \leqslant i \leqslant n$, either

(a) $F_i \in Ax$, or
(b) There exist $j_1, \cdots, j_m$, with $1 \leq j_k < i$ for $k = 1, \cdots, m$, such that

$$\frac{F_{j_1}, \cdots, F_{j_m}}{F_i} \in Pr$$

Expressed in words, $\vdash_{Ax,Pr} F$ if and only if there is a finite sequence of correctness formulae $F_1, \cdots F_n$ having F as its last element, such that each element of the sequence is either an axiom or is the conclusion of an inference which has its premises in the formulae occurring to the left of it in the sequence. In practice when we present a proof we list the premises of the proof from first to last. Any premise which is the conclusion of application of an inference will have an explanation of its proof listed. In addition, we permit formulae which have been previously proved to appear in proofs as axioms. This permits us to prove formulae in a modular fashion.

We want to use this method of formal proof to capture the interesting properties of the correctness formulae we have been developing. We can now formally state what we mean by soundness and completeness in a formal proof system. A proof system Ax, Pr is said to be *sound* if

$$\vdash_{Ax,Pr} F \supset \models F$$

and *complete* if

$$\models F \supset \vdash_{Ax,Pr} F$$

Since we are interested in correctness formulae F of the form $\{P\}K\{Q\}$ having to do with Hg programs we need a system Ax, Pr such that for all $P, Q \in Assn$ and $K \in Code$,

$$\vdash_{Ax,Pr} \{P\}K\{Q\} \text{ iff } \models \{P\}K\{Q\}$$

We present such a system for the language Hg without procedure calls. The assignment axiom and proof rules are actually schema for developing correct axioms and proof rules. A particular axiom or rule is generated by substituting the selectors, expressions,

assertions, and statements of interest.

Definition 4–4: (formal proof system for Hg without procedure calls)
(1) The set of axioms Ax consists of
  (a) All valid assertions
  (b) $\{\neg eq(s,\perp) \wedge \Delta_s^e(P)\}s := e\{P\}$
(2) The set of proof rules Pr consists of

(a) $$\frac{\{P_1\}k\{P_2\},\{P_2\}K\{P_3\}}{\{P_1\}k;K\{P_3\}}$$

(b) $$\frac{\{P \wedge b\}K_1\{Q\},\ \{P \wedge \neg b\}K_2\{Q\}}{\{P \wedge def(b)\}\ \textbf{if}\ b\ \textbf{then}\ K_1\ \textbf{else}\ K_2\ \textbf{endif}\ \{Q\}}$$

(c) $$\frac{\{P \supset P_1\},\{P_1\}K\{Q_1\},\{Q_1 \supset Q\}}{\{P\}K\{Q\}}$$

(d) $$\frac{\{P \wedge b\}K\{P \wedge def(b)\}}{\{P \wedge def(b)\}\ \textbf{while}\ b\ \textbf{loop}\ K\ \textbf{endloop}\ \{P \wedge \neg b\}}$$

Theorem 4–6: Soundness and completeness of Ax, Pr
Let Ax and Pr be as in Definition 4–4. Then for all $P, Q \in Assn$, $K \in Code$ with K not including a statement of the form $p(\bar{\alpha})$,

$$\vdash_{Ax,Pr} \{P\}K\{Q\} \text{ iff } \models \{P\}K\{Q\}$$

Proof:

Only If: We show that each formally provable F of the form $\{P\}K\{Q\}$ is valid. Let $F_1,\cdots,F_n$, with $F_n \equiv F$ be a formal proof of F. We show that for $i=1,\cdots,n$, $\models F_i$, therefore $\models F$. The proof proceeds by induction, showing F_1 is valid, and if for each $i, 1<i\leqslant n$, if $\models F_1,\cdots \models F_{i-1}$ then $\models F_i$.

Basis:
Clearly F_1 is an axiom, therefore by the definition of Ax it is either a valid assertion or it has the form $\{\neg eq(s,\perp) \wedge \Delta_s^e(P)\}s := e\{P\}$, which is valid by Theorem 3–6.

Induction:
Assuming $F_1,\cdots,F_{i-1}$ are all valid, we show that F_i is valid. If F_i is an axiom, its validity follows as in the basis case. Otherwise $\exists f_{j_1},\cdots,f_{j_m}$, $1\leqslant j_k<i$ for $k=1,\cdots,m$, such that $\dfrac{F_{j_1},\cdots,F_{j_m}}{F_i} \in Pr$, where Pr consists of the inference rules from Definition 4–4. By induction, $F_{j_1},\cdots,F_{j_m}$ are all

valid, and since the inferences in Pr are all sound (by Lemmas 4–1 to 4–5), the conclusion F_i follows.

If:

Assume that $\models \{P\}K\{Q\}$. We show that $\vdash_{Ax,Pr} \{P\}K\{Q\}$ by induction on the complexity of K.

(1) $K \equiv s := e$.

We have $\models \{P\}s := e\{Q\}$, therefore by Corollary 3–7

$$\models \{P \supset \neg eq(s,\perp) \wedge \Delta_s^e(Q)\}$$

Hence since by definition Ax includes all true formulae

$$\vdash_{Ax,Pr} \{P \supset \neg eq(s,\perp) \wedge \Delta_s^e(Q)\}$$

and again by definition of Ax,

$$\vdash_{Ax,Pr} \{\neg eq(s,\perp) \wedge \Delta_s^e(Q)\}s := e\{Q\}$$

and clearly we have

$$\vdash_{Ax,Pr} \{Q \supset Q\}$$

and the desired result

$$\vdash_{Ax,Pr} \{P\}s := e\{Q\}$$

follows by the rule of consequence.

(2) $K \equiv K_1;K_2$

Since $\models \{P\}K_1;K_2\{Q\}$, it is clear that

$$\models \{P\}K_1\{wp(K_2,Q)\}$$

since were this not the case, there is a state σ' resulting from execution of K_1 in a state σ satisfying P, such that execution of K_2 in σ' does not result in a state in which Q holds. It is also clear from the definition of wp that

$$\models \{wp(K_2,Q)\}K_2\{Q\}$$

By assumption of expressiveness of our assertion language $Assn$, there is an $R \in Assn$, such that

$$\models \{R = wp(K_2,Q)\}$$

Thus we have

$$\models \{P\}K_1\{R\} \text{ and } \models \{R\}K_2\{Q\}$$

and by induction

$$\vdash_{Ax,Pr} \{P\}K_1\{R\} \text{ and } \vdash_{Ax,Pr} \{R\}K_2\{Q\}$$

And our desired result

$$\vdash_{Ax,Pr} \{P\}K_1;K_2\{Q\}$$

follows from the rule of composition.

(3) $K \equiv$ **if** b **then** K_1 **else** K_2 **endif**

By Corollary 4–3, from $\models \{P \wedge def(b)\}$ **if** b **then** K_1 **else** K_2 **endif** $\{Q\}$ we can deduce that

$$\models \{P \wedge b\}K_1\{Q\} \text{ and } \models \{P \wedge \neg b\}K_2\{Q\}.$$

The result follows by induction together with the rule of conditionals.

(4) $K \equiv$ **while** b **loop** K_1 **endloop**

Let R be an assertion such that $\models \{R = wp(K_1,Q)\}$. (Our assumption of ex-

pressiveness of *Assn* guarantees that such an R exists.) Note that

$$\models \{R = wp(\textbf{while } b \textbf{ loop } K_1 \textbf{ endloop }, Q)$$
$$= wp(\textbf{if } b \textbf{ then } K_1; \textbf{while } b \textbf{ loop } K \textbf{ endloop } \textbf{else } \epsilon \textbf{ endif }, Q)\}$$

therefore we have

$$\models \{R \wedge b \supset wp(K_1; \textbf{while } b \textbf{ loop } K_1 \textbf{ endloop }, Q)$$
$$\supset wp(K_1, wp(\textbf{while } b \textbf{ loop } K_1 \textbf{ endloop }, Q))$$
$$\supset wp(K_1, R \wedge def(b))\}$$

So we know that

(a) $\models \{R \wedge b\} K_1 \{R \wedge def(b)\}$

By definition of R, we know that

(b) $\models \{P \supset R\}$

And by an argument similar to that demonstrating (a), above, we know that

(c) $\models \{R \wedge \neg b \supset Q\}$

By induction, from (a) we can see that

$$\vdash_{Ax, Pr} \{R \wedge b\} K_1 \{R \wedge def(b)\}$$

From this and the while rule we infer

$$\vdash_{Ax, Pr} \{R\} \textbf{ while } b \textbf{ loop } K_1 \textbf{ endloop } \{R \wedge \neg b\}$$

From (b) and (c) and the definition of Ax we see that

$$\vdash_{Ax, Pr} \{P \supset R\}$$

and

$$\vdash_{Ax, Pr} \{R \wedge \neg b \supset Q\}$$

And our result follows immediately from the rule of consequence.

□

4.2. Example program proof

Now that we have provided an assignment axiom and inference rules for each Hg statement aside from procedure call, we are ready to prove properties of simple programs. First consider a division program. This program is identical to the divide procedure in Example 2–1.

```
program
local-state : { g1:n1:[ # ]
              - x -> n2:[ ? ]
              - y -> n3:[ ? ]
              - q -> n4:[ ? ]
              - r -> n5:[ ? ]
              }

begin
        /q := 0;
        /r := /x;
        while /r ≥ /y loop
                /r := /r - /y;
                /q := /q + 1;
        endloop
end
```

The program above, for which we have left the initial values unspecified, divides the value of $/x$ by $/y$ and stores the quotient in $/q$ and remainder in $/r$. Although, the idea is simple to understand, a pre– and post–condition which capture this idea are not trivial to develop. A meaningful assertion about a divide program must not only assert that the quotient times divisor plus remainder equals the dividend, but that in perform–ing the division, the divisor and dividend do not change. Consider the precondition

$$/x=X \ \wedge \ /y=Y \ \wedge \ X \neq \perp \ \wedge \ Y \neq \perp$$

We prove that if this precondition holds in the initial state of the program above, then the following postcondition will hold in the state following termination of the pro–gram:

$$/x=X \ \wedge \ /y=Y \ \wedge /x=/y \times /q \ + \ /r \ \wedge \ \neg(/r \geq /y).$$

We proceed with the proof in a backward fashion, starting with our postcondition and working toward the precondition. We first determine a precondition for the while loop which will result in the truth of the postcondition above. Using the notation of Lemma 4–5, we let

$$P \equiv (/x=X \ \wedge \ /y=Y \ \wedge \ /x=/y \times /q \ + \ /r)$$

and

$$b \equiv /r \geq /y$$

Hence we must show that

$$\{/x=X \wedge /y=Y \wedge /x=/y\times/q + /r \wedge /r \geqslant /y\}$$
$$/r := /r - /y;$$
$$/q := /q + 1$$
$$\{/x=X \wedge /y=Y \wedge /x=/y\times/q + /r \wedge def\,(/r \geqslant /y)\} \qquad (*)$$

Once again, proceeding backwards we determine the precondition for the statement $/q := /q + 1$ which results in the postcondition (*). To do this, we apply $\Delta_{/q}^{/q\,+\,1}$, which results in $wp(/q := /q + 1,(*))$, which is

$$(eq(/q,/x) \wedge /q + 1=X \vee \neg eq(/q,/x) \wedge /x=X) \qquad (**)$$

$$\wedge\,(eq(/q,/y) \wedge /q + 1=Y \vee \neg eq(/q,/y) \wedge /y=Y)$$
$$\wedge \exists_{z_1,z_2,z_3,z_4,z_5}(z_2 + z_3=z_1 \wedge z_4\times z_5=z_2$$
$$\wedge(eq(/x,/q) \wedge /q + 1=z_1 \vee \neg eq(/x,/q) \wedge /x=z_1)$$
$$\wedge\,(eq(/y,/q) \wedge /q + 1=z_4 \vee \neg eq(/y,/q) \wedge /y=z_4)$$
$$\wedge\,(eq(/q,/q) \wedge /q + 1=z_5 \vee \neg eq(/q,/q) \wedge /q=z_5)$$
$$\wedge\,(eq(/r,/q) \wedge /q + 1=z_3 \vee \neg eq(/r,/q) \wedge /y=z_3))$$

$$\wedge \exists_{z_6,z_7}(\neg(z_6 \geqslant z_7) \vee (z_6 \geqslant z_7)$$
$$\wedge(eq(/r,/q) \wedge /q + 1=z_6 \vee \neg eq(/r,/q) \wedge /r=z_6)$$
$$\wedge(eq(/y,/q) \wedge /q + 1=z_7 \vee \neg eq(/r,/q) \wedge /y=z_7))$$

This precondition is quite unwieldy and application of Δ to this condition would be quite tedious. Fortunately by definition of *Stat* we have:

$$\models \{/x=X \wedge /y=Y \wedge /x=/y\times(/q + 1) + /r \wedge def\,(/r \geqslant /y) \supset (**)\}$$

So application of the rule of consequence gives us a formal proof of the formula:

$$\{/x=X \wedge /y=Y \wedge /x=/y\times(/q + 1) + /r \wedge def\,(/r \geqslant /y)\}$$
$$/q := /q + 1$$
$$\{/x=X \wedge /y=Y \wedge /x=/y\times/q + /r \wedge def\,(/r \geqslant /y)\}$$

Now we must determine a precondition for the statement $/r := /r - /y$ for which the new postcondition above holds. $\Delta_{/r}^{/r\,-\,/y}$ gives us the correct precondition,

$wp(/r := /r - /y,(**))$, which we will not expand here. The reader can verify the truth of the formula:

$$\{/x=X \wedge /y=Y \wedge /x=/y\times(/q + 1) + (/r - /y) \wedge def\ ((/r - /y)\geqslant /y) \supset$$
$$wp(/r := /r - /y,(**))\}$$

We have now constructed a proof of

$$\{/x=X \wedge /y=Y \wedge /x=/y\times(/q + 1) + (/r - /y) \wedge def\ ((/r - /y)\geqslant /y)\}$$
$$/r := /r - /y$$
$$\{/x=X \wedge /y=Y \wedge /x=/y\times(/q + 1) + /r \wedge def\ (/r \geqslant /y)\}$$

We must appeal to the rule of consequence once more. Our result will be true only if the following condition holds in the underlying logic:

$$/x=/y\times/q + /r \supset$$
$$/x=/y\times(/q + 1) + (/r - /y)$$

that is, we must know that multiplication is equivalent to repeated addition in our underlying logic. If the underlying logical model is such that this condition does not hold, then the proof of this program may be impossible to develop. In fact, the program may not even be correct at all. It is reasonable to assume that one would choose a logical model in which the functions + and - had interpretations which would make this statement true. We assume this condition is an axiom of the underlying logic.

We can also make use of the following implication:

$$/r \geqslant /y \supset def\ ((/r - /y)\geqslant /y)$$

The truth of this is apparent, since if $/r \geqslant /y$ holds, then selectors $/r$ and $/y$ are defined and thus $(/r - /y)\geqslant /y$ must be defined as well.

We have now shown all the steps necessary to apply the while condition, and the precondition for the while statement is

$$\{/x=X \wedge /y=Y \wedge /x=/y\times/q + /r \wedge def\ ((/r - /y)\geqslant /y)\} \qquad (***)$$

Applying $\Delta^{/x}_{/r}$ gives us $wp(/r := /x,(***))$ and the truth of the following formula follows from the definition of *Stat*:

$$\{/x=X \wedge /y=Y \wedge /x=/y\times/q + /x \wedge def\,((/x - /y)\geqslant/y) \supset wp(/r := /x,(***))\}$$

Applying $\Delta^{0}_{/q}$ to

$$\{/x=X \wedge /y=Y \wedge /x=/y\times/q + /x \wedge def\,((/x - /y)\geqslant/y) \qquad (****)$$

gives us $wp(/q := 0,(****))$, and we can show that

$$\{/x=X \wedge /y=Y \wedge /x=/y\times0 + /x \wedge def\,((/x - /y)\geqslant/y) \supset wp(/q := 0,(****))\}$$

Once again, using the rule of consequence, we know that if the following condition holds

$$\{/x=X \wedge /y=Y \wedge def\,((/x - /y)\geqslant/y)\supset$$
$$/x=X \wedge /y=Y \wedge /x=/y\times0 + /x \wedge def\,((/x - /y)\geqslant/y)\}$$

and since it is clear that

$$X\neq\perp \wedge Y\neq\perp \supset def\,((/x - /y)\geqslant/y)$$

then the truth of the precondition

$$\{/x=X \wedge /y=Y \wedge X\neq\perp \wedge Y\neq\perp\}$$

in the initial state of the above program, implies the truth of the postcondition

$$\{/x=X \wedge /y=Y \wedge /x=/y\times/q + /r \wedge \neg(/r\geqslant/y)\}.$$

We recap the formal proof here:

Lemma 4–7: While loop formula:

$$\vdash_{Ax,Pr}$$

$\{/x=X \wedge /y=Y \wedge /x=/y\times/q + /r \wedge def\,(/r\geqslant/y)\}$
while $/r\geqslant/y$ **loop**
  $/r := /r - /y;$
  $/q := /q + 1;$
endloop
$\{/x=X \wedge /y=Y \wedge /x=/y\times/q + /r \wedge \neg(/r\geqslant/y)\}$

Proof:

$$\{/x=X \wedge /y=Y \wedge /x=/y\times/q + /r \wedge /r\geqslant/y \supset$$
$$/x=X \wedge /y=Y \wedge /x=/y\times(/q + 1) + (/r - /y) \wedge def\,((/r - /y)\geqslant/y)\}$$

Axiom of the underlying logic.

$$\{/x = X \wedge /y = Y \wedge /x = /y \times (/q + 1) + (/r - /y) \wedge def\,((/r - /y) \geqslant /y)\}$$
$$/r := /r - /y$$
$$\{/x = X \wedge /y = Y \wedge /x = /y \times (/q + 1) + /r \wedge def\,(/r \geqslant /y)\}$$

Assignment axiom and the rule of consequence.

$$\{/x = X \wedge /y = Y \wedge /x = /y \times (/q + 1) + /r \wedge def\,(/r \geqslant /y)\}$$
$$/q := /q + 1$$
$$\{/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge def\,(/r \geqslant /y)\}$$

Assignment axiom and the rule of consequence.

$$\{/x = X \wedge /y = Y \wedge /x = /y \times (/q + 1) + (/r - /y) \wedge def\,((/r - /y) \geqslant /y)\}$$
$$/r := /r - /y;$$
$$/q := /q + 1$$
$$\{/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge def\,(/r \geqslant /y)\}$$

Sequential composition.

$$\{/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge /r \geqslant /y\}$$
$$/r := /r - /y;$$
$$/q := /q + 1$$
$$\{/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge def\,(/r \geqslant /y)\}$$

Rule of consequence.

$$\{/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge def\,(/r \geqslant /y)\}$$
while $/r \geqslant /y$ **loop**
$/r := /r - /y;$
$/q := /q + 1;$
endloop
$$/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge \neg(/r \geqslant /y)$$

While rule.

□

Theorem 4–8: Division program formula:

$$\vdash_{Ax,Pr} \begin{array}{l} \{/x = X \wedge /y = Y \wedge X \neq \perp \wedge Y \neq \perp\} \\ \quad /q := 0; \\ \quad /r := /x; \\ \quad \textbf{while } /r \geqslant /y \textbf{ loop} \\ \quad\quad /r := /r - /y; \\ \quad\quad /q := /q + 1; \\ \quad \textbf{endloop} \\ \{/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge \neg(/r \geqslant /y)\} \end{array}$$

Proof:

$\{/x = X \wedge /y = Y \wedge X \neq \perp \wedge Y \neq \perp \supset /x = X \wedge /y = Y \wedge /x = /y \times 0 + /x \wedge def\,((/x - /y) \geqslant /y)\}$

Axiom of the underlying logic.

$$\{/x = X \wedge /y = Y \wedge /x = /y \times 0 + /x \wedge def\,((/x - /y) \geqslant /y)\}$$

$$/q := 0$$

$$\{/x = X \wedge /y = Y \wedge /x = /y \times /q + /x \wedge def\,((/x - /y) \geqslant /y)\}$$

Assignment axiom and the rule of consequence.

$$\{/x = X \wedge /y = Y \wedge /x = /y \times /q + /x \wedge def\,((/x - /y) \geqslant /y)\}$$

$$/r := /x$$

$$\{/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge def\,((/r - /y) \geqslant /y)\}$$

Assignment axiom and the rule of consequence.

$$\{/x = X \wedge /y = Y \wedge /x = /y \times 0 + /x \wedge def\,((/x - /y) \geqslant /y)\}$$

$$\begin{array}{l} /q := 0; \\ /r := /x \end{array}$$

$$\{/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge def\,((/r - /y) \geqslant /y)\}$$

Sequential composition.

$$/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge def\,((/r - /y) \geqslant /y) \supset$$

$$/x = X \wedge /y = Y \wedge /x = /y \times /q + /r \wedge def\,(/r \geqslant /y)$$

Axiom of the underlying logic.

$$\{/x=X \wedge /y=Y \wedge X \neq \perp \wedge Y \neq \perp \supset$$
$$/x=X \wedge /y=Y \wedge /x=/y \times 0 + /x \wedge def\,((/x - /y) \geqslant /y)\}$$

Axiom of the underlying logic.

$$\{/x=X \wedge /y=Y \wedge X \neq \perp \wedge Y \neq \perp\}$$

$$/q := 0;$$
$$/r := /x$$
$$\{/x=X \wedge /y=Y \wedge /x=/y \times /q + /r \wedge def\,(/r \geqslant /y)\}$$

Rule of consequence.

$$\{/x=X \wedge /y=Y \wedge X \neq \perp \wedge Y \neq \perp\}$$
$$/q := 0;$$
$$/r := /x;$$
$$\textbf{while } /r \geqslant /y \textbf{ loop}$$
$$/r := /r - /y;$$
$$/q := /q + 1;$$
$$\textbf{endloop}$$
$$/x=X \wedge /y=Y \wedge /x=/y \times /q + /r \wedge \neg(/r \geqslant /y)$$

Lemma 4–7 and Rule of composition.

We have now developed a proof of a meaningful program. The complexity of the proof in this system should be obvious. The generality of the data structures requires that the preconditions of assignments be complex. We were able to remove some of this complexity because single graph selectors selecting nodes in the root graph of a state are neither aliased with each other nor components of structures. This averted the need to apply Δ to increasingly complex conditions.

4.3. A Restricted Procedure Call Rule

In this section we present an inference rule which can be used to prove properties of a class of procedure calls. We restrict our attention to non–recursive procedures in programs where no aliases occur within the actual parameters of any procedure call. This

restriction on aliasing permits us to use the simpler substitution rule ∇, rather than the assignment substitution rule Δ, in the definition of the procedure call rule. We must, however, first define the extension of ∇ to tuples.

To extend ∇_s^r to vectors, we define $\nabla_{\bar{s}}^{\bar{r}}(P)$ for vectors $\bar{s}$ and $\bar{r}$ to be the simultaneous substitution of $\bar{r}$ for $\bar{s}$ in assertion P using the substitution defined by ∇. We only apply $\nabla_{\bar{s}}^{\bar{r}}$ in cases where the structure of $\bar{r}$ and $\bar{s}$ guarantee that this captures the meaning of assignment of the values of $\bar{r}$ to selectors of $\bar{s}$.

Proof of the soundness and completeness of the procedure call rule requires several preliminary definitions and results. The first definition, the *non-prefix property* on a vector of selectors, is a key condition for defining actual and formal parameter lists in procedure calls so that transmission of one argument or result does not interfere with transmission of another.

Definition 4-5: (The non-prefix property)
A tuple of selectors $\bar{s}$ is said to display the non-prefix property in state σ, written *nonprefix*$(\bar{s})(\sigma)$ if the prefixes of selectors in $\bar{s}$ are pairwise neither identical nor aliased. That is, if given $|\bar{s}|=n$, then

$$\forall i,1\leqslant i\leqslant n,\ \forall j,1\leqslant j\leqslant n, j\neq i,\ \forall r[prefix(r,s\downarrow i)\Rightarrow [\![\neg eq(r,s\downarrow j)]\!](\sigma)]$$

$$\wedge\ \forall i,1\leqslant i\leqslant n,\forall r[prefix(r,s\downarrow i)\wedge r\not\equiv \bar{s}\downarrow i\Rightarrow [\![\neg eq(r,\bar{s}\downarrow i)]\!](\sigma)]$$

If $\forall\sigma\in Stat$, *nonprefix*$(\bar{s})(\sigma)$, we write *nonprefix*$(\bar{s})$.

Definition 4-6: (*selectors*, selector set function)
The function *selectors*$:Assn\cup Code\rightarrow 2^{Sel}$ when presented with an assertion P or code segment K returns the set of all selectors appearing in the argument assertion P or code segment K.

Lemma 4-9: Let $h_1=\langle G_1,V_1,r_1\rangle$ and $h_2=\langle G_2,V_2,r_2\rangle$ be two h-graphs, and $n_1\in nodeset(h_1), n_2\in nodeset(h_2)$. If $V_1^+(n_1)\equiv V_2^+(n_2)$ then if $n\in nodeset(V_1^+)$
$V_1(n)=V_2(n)$

Proof: Immediate from the definition of V^+ (Definition 2-7).

□

The following lemma states that if there are two states σ_1 and σ_2, and two tuples of selectors $\bar{\eta}$ and $\bar{\alpha}$ of the same length satisfying $nonprefix(\bar{\eta})(\sigma_1)$ and $nonprefix(\bar{\alpha})(\sigma_2)$ then if the sub-h-graphs defined by selectors of $\bar{\eta}$ and $\bar{\alpha}$ are pairwise identical on σ_1 and σ_2 respectively, and there is some assertion P involving only selectors with prefixes in $\bar{\eta}$, then $[\![\nabla_{\bar{\eta}}^{\bar{\alpha}}(P)]\!](\sigma_2) \Leftrightarrow [\![P]\!](\sigma_1)$. Think of $\bar{\eta}$ as the formal parameter selectors of some procedure and $\bar{\alpha}$ as the selectors appearing in some call of that procedure. $[\![P]\!](\sigma_1)$ is a property known about the procedure state σ_1 at the start of execution of the body of the procedure. $[\![\nabla_{\bar{\eta}}^{\bar{\alpha}}(P)]\!](\sigma_2)$ is the same property stated in terms of the actual parameters in the calling state σ_2.

Lemma 4-10: If $\sigma_1 \equiv \langle G_1, V_1, r_1 \rangle$ and $\sigma_2 \equiv \langle G_2, V_2, r_2 \rangle$ and

$$\exists \bar{\eta}, \bar{\alpha} \in Sel^N, |\bar{\eta}| = |\bar{\alpha}| = n, nonprefix(\bar{\eta})(\sigma_1) \wedge nonprefix(\bar{\alpha})(\sigma_2)$$
$$\wedge \forall i, 1 \leq i \leq n, V_1^{\dagger}([\![\bar{\eta}{\downarrow}i]\!](\sigma_1)) \equiv V_2^{\dagger}([\![\bar{\alpha}{\downarrow}i]\!](\sigma_2))$$

then for any assertion Q satisfying the restriction that all selectors in Q have prefixes in $\bar{\eta}$, that is $\forall s \in selectors(Q), \exists i, 1 \leq i \leq |\bar{\eta}|, prefix(\bar{\eta}{\downarrow}i, s)$

$$[\![\nabla_{\bar{\eta}}^{\bar{\alpha}}(Q)]\!](\sigma_2) \Leftrightarrow [\![Q]\!](\sigma_1)$$

Proof: By (simultaneous) induction on the cases of ∇.

Case 1: $Q \equiv \exists x(P)$

$$[\![\nabla_{\bar{\eta}}^{\bar{\alpha}}(\exists x(P))]\!](\sigma_2) = [\![\exists x(\nabla_{\bar{\eta}}^{\bar{\alpha}}(P))]\!](\sigma_2)$$

(definition of ∇)

$$\Leftrightarrow \exists x([\![\nabla_{\bar{\eta}}^{\bar{\alpha}}(P)]\!](\sigma_2))$$

(immediate from definition 1-6)

$$\Leftrightarrow \exists x([\![P]\!](\sigma_1))$$

(by induction)

$$\Leftrightarrow [\![\exists x(P)]\!](\sigma_1)$$

Cases 2 through 6 are essentially identical to Case 1.

Case 7: since $[\![\nabla_{\bar{\eta}}^{\bar{\alpha}}(d_1 = d_2)]\!](\sigma_2) = [\![\nabla_{\bar{\eta}}^{\bar{\alpha}}(d_1) = \nabla_{\bar{\eta}}^{\bar{\alpha}}(d_2)]\!](\sigma_2) \Leftrightarrow [\![d_1 = d_2]\!](\sigma_1)$ will clearly hold if $R(\nabla_{\bar{\eta}}^{\bar{\alpha}}(d_1))(\sigma_2) = R(d_1)(\sigma_1)$ and $R(\nabla_{\bar{\eta}}^{\bar{\alpha}}(d_2))(\sigma_2) = R(d_2)(\sigma_1)$, we need merely show that this holds for all the cases of d_1 and d_2.

We show the proof for the case of d being a selector u:

We know by our assumptions that there exist unique i and n, $n \leq m$, such that

$$\bar{\eta}\downarrow i \equiv u_1 \cdots u_n \tag{1}$$

This makes it easy to show the following:

$$R(\nabla_{\bar{\eta}}^{\bar{\alpha}}(u))(\sigma_2) = V_2([\![\nabla_{\bar{\eta}}^{\bar{\alpha}}(u_1 \cdots u_n u_{n+1} \cdots u_m)]\!](\sigma_2)))$$

$$= V_2([\![r_1 \cdots r_k u_{n+1} \cdots u_m]\!](\sigma_2))$$

$$= V_2([\![u_{n+1} \cdots u_m]\!](V_2^+([\![r_1 \cdots r_k]\!](\sigma_2))))$$

By Lemma 2–2.

$$= V_1([\![u_{n+1} \cdots u_m]\!](V_2^+([\![r_1 \cdots r_k]\!](\sigma_2))))$$

By Lemma 4–9.

$$= V_1([\![u_{n+1} \cdots u_m]\!](V_1^+([\![u_1 \cdots u_n]\!](\sigma_1))))$$

Assumption.

$$= V_1([\![u_1 \cdots u_n u_{n+1} \cdots u_m]\!](\sigma_1))$$

$$= V_1([\![u]\!](\sigma_1))$$

$$= R(u)(\sigma_1)$$

or restated,

$$R(\nabla_{\bar{\eta}}^{\bar{\alpha}}(u))(\sigma_2) = R(u)(\sigma_1)$$

A corresponding argument can be made for the other cases of ∇.

□

Now we present a lemma which relates the behavior of ∇ to the assignment of parameter values into and out of procedure calls. We actually prove a slightly more general result, that is if one state, σ, has a value function and graphset which are subsets of the value function and graphset of a different state with a different root graph, σ', and there are two identical length tuples of selectors, $\bar{\alpha}$ defined on σ and $\bar{\eta}$ defined on σ', satisfying the nonprefix property in their respective states, then substituting $\bar{\alpha}$ for $\bar{\eta}$ in an assertion P, all the selectors of which have prefixes in $\bar{\eta}$, yields an assertion which when evaluated in σ has the same value as P when evaluated in $\sigma'\{R(\bar{\alpha})(\sigma):\bar{\eta}\})$, that is, the state σ' with the values of $\bar{\alpha}$ in σ assigned to the nodes selected by $\bar{\eta}$.

Lemma 4-11: Given h-graphs $\sigma = \langle G,V,r\rangle$, $\sigma' = \langle G',V',r'\rangle$, such that

$$V'=V \cup V'',\ domain(V)\cap domain(V'') = \emptyset \text{ and}$$
$$G'=G \cup G'',\ nodeset(G)\cap nodeset(G'') = \emptyset,$$

two tuples of selectors $\bar{\alpha},\bar{\eta}\in Sel^N$, with $|\bar{\alpha}| = |\bar{\eta}| = n$, and assertion P such that

$$\forall_{s\in selectors(P)},\exists i,1\leqslant i\leqslant n, prefix(\bar{\eta}\downarrow i,s) \text{ and}$$
$$nonprefix(\bar{\alpha})(\sigma) \wedge nonprefix(\bar{\eta})(\sigma').$$

Then

$$[\![\nabla_{\bar{\eta}}^{\bar{\alpha}}(P)]\!](\sigma) \Leftrightarrow [\![P]\!](\sigma'\{R(\bar{\alpha})(\sigma):\bar{\eta}\})$$

Proof: We show that the conditions above guarantee that

$$\forall i,1\leqslant i\leqslant n,\ V^+([\![\bar{\alpha}\downarrow i]\!](\sigma)) \equiv V'\{R(\bar{\alpha})(\sigma):\bar{\eta})\}^+([\![\bar{\eta}\downarrow i]\!](\sigma'\{R(\bar{\alpha})(\sigma):\bar{\eta})\}))$$

then apply the preceding theorem to obtain our desired result.

Notice that by definition of variant of state, since the nonprefix property holds for both $\bar{\alpha}$ and $\bar{\eta}$,

$$\forall i,1\leqslant i\leqslant n, V([\![\bar{\alpha}\downarrow i]\!](\sigma)) \equiv V'\{R(\bar{\alpha})(\sigma):\bar{\eta})\}([\![\bar{\eta}\downarrow i]\!](\sigma'\{R(\bar{\alpha})(\sigma):\bar{\eta})\})).$$

Now if $V([\![\bar{\alpha}\downarrow i]\!](\sigma))\in\Delta$, then clearly

$$V^+([\![\bar{\alpha}\downarrow i]\!](\sigma)) = V'\{R(\bar{\alpha}\downarrow i)(\sigma):\bar{\eta}\downarrow i)\}^+([\![\bar{\eta}\downarrow i]\!](\sigma'\{R(\bar{\alpha})(\sigma):\bar{\eta})\})).$$

On the other hand, if $V([\![\bar{\alpha}\downarrow i]\!](\sigma))\in G$, then since V and $V'\{R(\bar{\alpha}\downarrow i)(\sigma):\bar{\eta}\downarrow i)\}$ agree on $nodeset(G)$,

$$V^+([\![\bar{\alpha}\downarrow i]\!](\sigma)) = V'\{R(\bar{\alpha}\downarrow i)(\sigma):\bar{\eta}\downarrow i)\}^+([\![\bar{\eta}\downarrow i]\!](\sigma'\{R(\bar{\alpha})(\sigma):\bar{\eta})\})),$$

and our result follows from the preceding lemma.

□

We can now present the (non-recursive) procedure call rule and its proof. The form of this rule is suggested in Oppen and Cartwright [1] who develop a similar rule for a Pascal subset. The presentation is based on a much different substitution rule as there is no demarcation between semantics of the language and the underlying logic as there is here. A recursion hypothesis and a rule for recursive procedure calls following the presentation of Oppen and Cartwright could certainly be developed, but we omit such a presentation here.

The appearance of nested implications in the procedure call rule permits the pre- and post-conditions on the procedure body to be related to the pre- and post-conditions of the call. A simpler rule which tries to apply the same pre-conditions to the body as to

the call cannot permit assertions involving selectors with prefixes different from the argument selectors of the procedure call to be verified in the context of a call.

The procedure call rule states that if the following conditions hold:

(1) procedure p has initial state σ_p, body K_p, and formal parameters $\bar{\eta}_p$ in the procedure map π,

(2) P and Q are valid pre– and post–conditions respectively of the body of procedure p,

(3) all the selectors in P and Q have prefixes in the formal parameters of the procedure, and

(4) if that P with the formals replaced by the actuals implies Q with the formals replaced by some variables in turn implies that R implies S with the actuals replaced by those same variables,

then indeed R and S are pre– and post–conditions respectively of a call of procedure p in a state where the actual parameters satisfy the non–prefix property.

We use a version of ∇ which is a slight extension over the rule presented in Chapter 3. This extension lets us substitute variables for selectors in the logical implication forming the second premise of the procedure call rule. Where these variables are substituted for full selectors, the resulting formula may be evaluated directly in the underlying logic by substituting a constant with the same value as the variable for the selector. But when a variable is substituted for a prefix of a selector, the intent is to apply the remainder of the selector to the value which has been substituted for the prefix. If such a variable appears in a formula, its value is to be bound before the formula is converted into a wff of the underlying logic and any use of these variables within a single formula is universally quantified. For example, if the value v is bound to variable x and x is substituted for selector s, in the case of a selector $s' = su$, the result of the substitution, written $\dagger(x,u)$ (as seen in the example of the next section) would be evaluated in h–graph $h=\langle G,V,r\rangle$ by applying the selector u to the sub–h–graph of h defined by the value v,

that is $[\![u]\!](V^{\perp}(v))$. This returns $\perp$ if x is bound to some value v which is not in the graphset of h.

Though we could formalize this substitution, we feel the reader should not be burdened with yet another subsitution rule. In practice when applying the procedure call rule one never actually evaluates a selector of the form $\uparrow(x,u)$; one merely shows that a formula involving such a selector is a tautology.

Lemma 4–12: (nonprefix procedure call rule)
Given $\pi(p) = \langle \sigma_p, K_p, \bar{\eta} \rangle$, $\bar{x}$ a vector of variables, $|\bar{x}|=|\bar{\eta}|$, and $P,Q \in Assn$ such that

$$\forall s \in selectors(P) \cup selectors(Q),\ \exists i, 1 \leqslant i \leqslant n,\ prefix(\bar{\eta}\downarrow i, s)$$

$$\frac{\{P\}K_p\{Q\},\quad \{(\nabla^{\bar{\alpha}}_{\bar{\eta}}(P) \supset \nabla^{\bar{x}}_{\bar{\eta}}(Q)) \supset (R \supset \nabla^{\bar{x}}_{\bar{\alpha}}(S))\}}{\{nonprefix(\bar{\alpha}) \wedge R\}\ p(\bar{\alpha})\ \{S\}}$$

Proof:
To prove the soundness of this rule, we must show that the conditions above imply the truth of $\{nonprefix(\bar{\alpha}) \wedge R\}\ p(\bar{\alpha})\ \{S\}$.

$$\{P\}K_p\{Q\} \Rightarrow \forall \sigma, [\![P]\!](\sigma) \supset [\![Q]\!](Last(Comp(K_p)(\sigma)))$$

Assume $\sigma = \langle G, V, r \rangle$ and σ' is as in the procedure call case of *Comp*, Definition 2–30. Then we know that

$$[\![\nabla^{\bar{\alpha}}_{\bar{\eta}}(P)]\!](\sigma) \supset [\![P]\!](\sigma'\{R(\bar{\alpha})(\sigma):\bar{\eta}\})$$

since we can apply Lemma 4–11. We will refer to $\sigma'\{R(\bar{\alpha})(\sigma):\bar{\eta}\}$ as σ'', as in Definition 2–30.

We are given that $[\![P]\!](\sigma'') \supset [\![Q]\!](Last(Comp(K_p)(\sigma'')))$.

If we choose values for $\bar{x}$ such that $[\![\bar{x} = \bar{\eta}]\!](Last(Comp(K_p)(\sigma'')))$ then we have

$$[\![Q]\!](Last(Comp(K_p)(\sigma''))) \supset [\![\nabla^{\bar{x}}_{\bar{\eta}}(Q)]\!](\sigma)$$

because Q contains only selectors with prefixes in $\bar{\eta}$.
Given this implication, we conclude from the second premise that

$$[\![R]\!](\sigma) \supset [\![\nabla^{\bar{x}}_{\bar{\alpha}}(S)]\!](\sigma),$$

where $\bar{x}$ is defined as above. Given $Last(Comp(K)(\sigma'')) = \langle G^*, V^*, r^* \rangle$, let $\sigma^* = \langle G^*, V^*, r \rangle$, r from σ, as in Definition 2–30. Then

$$[\![\nabla^{\bar{x}}_{\bar{\alpha}}(S)]\!](\sigma) \supset [\![S]\!](\sigma^*\{\bar{x}:\bar{\alpha}\})$$

by Lemma 4–11

$$= [\![S]\!](\sigma^*\{R(\bar{\eta})(Last(Comp(K_p)(\sigma'')):\bar{\alpha}\})$$
$$= [\![S]\!](Last(Comp(p(\bar{\alpha}))(\sigma))).$$

And since

$$[\![R]\!](\sigma) \supset [\![S]\!](Last(Comp(p(\bar{\alpha}))(\sigma))),$$

the rule is sound.

□

We can now extend the formal proof system for Hg, presented in Definition 4–4, to let us prove formulae concerning Hg programs with procedures satisfying the restrictions outlined above.

Definition 4–7: (formal proof system for Hg with procedure calls)
(i) The set of axioms Ax consists of precisely those axioms in Definition 4–4
(ii) The set of proof rules Pr consists of those proof rules in Definition 4–4, together with the rule
Given $\pi(p) = \langle \sigma_p, K_p, \bar{\eta} \rangle$, $\bar{x}$ a vector of variables, $|\bar{x}|=|\bar{\eta}|$, and $P, Q \in Assn$ such that

$$\frac{\begin{array}{c}\forall_{s \in selectors(P) \cup selectors(Q)}, \exists i, 1 \leqslant i \leqslant n, prefix(\bar{\eta}\downarrow i, s) \\ \{P\}K_p\{Q\}, \\ \{(\nabla_{\bar{\eta}}^{\bar{\alpha}}(P) \supset \nabla_{\bar{\eta}}^{\bar{x}}(Q)) \supset (R \supset \nabla_{\bar{\alpha}}^{\bar{x}}(S))\}\end{array}}{\{nonprefix(\bar{\alpha}) \wedge R\}\ p(\bar{\alpha})\ \{S\}}$$

We now establish the soundness and completeness of this axiom system.

Theorem 4–13: Let Ax and Pr be defined as in Definition 4–7. Then for all $P, Q \in Assn$, $K \in Code$,

$$\vdash_{Ax,Pr} \{P\}K\{Q\} \text{ iff } \models \{P\}K\{Q\}$$

Proof:

Only If: The validity of the axioms and soundness of the rules in Definition 4–7 are proved in Lemmas 4–1 through 4–5. The result follows by an argument exactly like that used in proof of Theorem 4–6.

If: Proof by induction on the complexity of K. If K is an assignment statement, a conditional statement, a while statement, or a sequence of statements, the argument is the same as in Theorem 4–6. Suppose $K \equiv p(\bar{\alpha})$. We must show that if $\models \{nonprefix(\bar{\alpha}) \wedge R\}\ p(\bar{\alpha})\ \{S\}$, then we can find assertions P and Q such that the conditions of the rule hold.

If we can show that for any program segment K and postcondition S, a true assertion of the form $\{R\}K\{S\}$ is always provable, then we know our call rule is complete.

Assume $p'(\bar{\alpha}')$ is a procedure call for which the rule is not complete and $p(\bar{\alpha})$ is the deepest call in the calling tree of p' for which the rule is not complete, with

$$\pi(p) = \langle \sigma, K, \bar{\eta} \rangle$$

Let S be an arbitrary postcondition for $p(\bar{\alpha})$. Let Q be the strongest postcondition for K given $P \equiv \bar{\eta}=\bar{x}$, that is $Q=sp(K,\bar{\eta}=\bar{x})$. By assumption, $\vdash_{Ax,Pr} \{P\}K\{Q\}$.

Let $P' = \left| \nabla_{\bar{x}}^{\bar{\alpha}} \nabla_{\bar{\eta}}^{\bar{y}} (Q) \supset \nabla_{\bar{\alpha}}^{\bar{y}} (S) \right|$.

It is easy to show that

$$\vdash_{Ax,Pr} \{R'\}p(\bar{\alpha})\{S\}$$

(The soundness proof suggests the method for this).

If R' is the weakest precondition of $p(\bar{\alpha})$ given S, that is $R'=wp(p(\bar{\alpha}),S)$ we have accomplished our goal, since any other true precondition will imply R'. We show by contradiction that this is the case.

Suppose R' is not the weakest precondition. Then

$$\exists \sigma \text{ s.t. } [\![\neg R']\!](\sigma) \wedge Last(Comp(p(\bar{\alpha}))(\sigma))=\perp,$$

or

$$[\![S]\!](Last(Comp(p(\bar{\alpha}))(\sigma))).$$

Let σ'' be as defined in procedure call alternative of $Comp$. We know that either

$$Last(Comp(K)(\sigma''))=\perp$$

or

$$[\![Q]\!](Last(Comp(K)(\sigma''))).$$

In the former case, $\nabla_{\bar{x}}^{\bar{\alpha}} \nabla_{\bar{\eta}}^{\bar{y}} (Q)$ is false for all $\bar{y}$. Hence R' is a vacuously true implication. In the latter case, $[\![\neg R']\!](\sigma)$ is true iff $[\![\nabla_{\bar{x}}^{\bar{\alpha}} \nabla_{\bar{\eta}}^{\bar{y}} (Q)]\!](\sigma)$ and $[\![\neg \nabla_{\bar{\alpha}}^{\bar{y}} (S)]\!](\sigma)$.

But $\nabla_{\bar{x}}^{\bar{\alpha}} \nabla_{\bar{\eta}}^{\bar{y}} (Q)$ is true only in those states with

$$[\![\bar{y}=\bar{\eta}]\!](Last(Comp(K)(\sigma''))).$$

And for such a state,

$$Last(Comp(p(\bar{\alpha}))(\sigma))=\sigma^*\{R(\bar{\eta})(Last(Comp(K)(\sigma''))):\bar{\alpha}\}$$

where σ^* is as defined in the procedure call alternative of $Comp$, and

$$[\![\nabla_{\bar{\alpha}}^{\bar{y}} (S)]\!](\sigma) = [\![S]\!](\sigma^*\{R(\bar{\eta})(Last(Comp(K)(\sigma''))):\bar{\alpha}\}),$$

since $[\![\bar{y} = \bar{\eta}]\!](Last(Comp(K)(\sigma')))$. And since the state on the right hand side of this expression is the final state in the procedure call, $[\![\nabla_{\bar{\alpha}}^{\bar{y}} (S)]\!](\sigma)$ is true, contrary to our assumption. Therefore $[\![R']\!](\sigma)$ is true and R' is the weakest precondition.

□

We now prove a theorem similar to Theorem 3–8 for procedure calls. This theorem states that if none of the selectors accessing nodes in the root graph of the program state is an alias of any other selector in that state, then executing a procedure call cannot change that fact. We can apply this theorem to help us reduce program proof complexity in the same way we did with Theorem 3–8.

Theorem 4–14: If $\sigma=\langle G,V,r\rangle$ and $[\![s]\!](\sigma)=n$, $n \in nodeset(r)$, and

$$\forall_{u_1,} \in Sel\sim\{s\}, [\![\neg eq(u_1,s)]\!](\sigma)$$

and if $\sigma'=Last(Comp(p(\bar{\alpha}))(\sigma))$ then

$$\forall_{u_1} \in Sel\sim\{s\}, [\![\neg eq(u_1,s)]\!](\sigma')$$

Proof: by contradiction:

Suppose $\exists\sigma \in Stat, s \in Sel$ such that

$$\exists_{u_1} \in Sel\sim\{s\}, [\![\neg eq(u_1,s)]\!](\sigma) \wedge [\![eq(u_1,s)]\!](Last(Comp(p(\bar{\alpha}))(\sigma)))$$

Then we should be able to find an assertion R which characterizes the state σ sufficiently well for the following formula to hold:

$$\{nonprefix(\bar{\alpha}) \wedge \neg(eq(u_1,s)) \wedge R\}p(\bar{\alpha})\{eq(u_1,s)\}$$

By the completeness of the procedure call rule, this means we should be able to find P and Q only involving selectors with prefixes in $\bar{\eta}$, such that

$$(\nabla_{\bar{\eta}}^{\bar{\alpha}}(P) \supset \nabla_{\bar{\eta}}^{\bar{x}}(Q)) \supset (\neg eq(u_1,s) \wedge R \supset eq(u_1,s))$$

is a valid formula.

This formula can never be satisfied, regardless of the form of P and Q.

□

4.4. Application of the Procedure Call Rule

Example 4–1: (Proof of Procedure Call *Push*)

In this section, we prove a property of a procedure call. The example is drawn from the stack program example of Chapter 2. The property we want to verify is that if the stack argument of the procedure call has value S before a call to *push*, and the value to be pushed is V, then in the state after the call, the stack's tail has value S and it's head has value V. We assume the following result:

Given $P = \langle \sigma, K, \pi \rangle$, $\pi(push) = \langle \sigma_{push}, K_{push}, \langle /stk, /value \rangle \rangle$, and

$$\models \quad \begin{array}{c} \{/value = V \ \wedge \ /stk = S\} \\ K_{push} \\ \{/stk\,/head = V \ \wedge \ /stk\,/tail = S\} \end{array}$$

Using the procedure call rule, with instance

$P = (/value = V \ \wedge \ /stk = S)$

$Q = (/stk\,/head = V \ \wedge \ /stk\,/tail = S)$

$R = (/value1 = V \ \wedge \ /stack = S)$

$S = (/stack\,/head = V \ \wedge \ /stack\,/tail = S)$

we can show that

$$\{nonprefix(\langle /stack, /value1 \rangle) \wedge /value1 = V \ \wedge \ /stack = S\}$$

$$push(\langle /stack, /value1 \rangle)$$

$$\{/stack\,/head = V \ \wedge \ /stack\,/tail = S\}$$

To do this, we must show that

$$\left(\nabla^{\langle /stack, /value1 \rangle}_{\langle /stk, /value \rangle}(/value = V \ \wedge \ /stk = S) \supset \nabla^{\langle x_1, x_2 \rangle}_{\langle /stk, /value \rangle}(/stk\,/head = V \ \wedge \ /stk\,/tail = S)\right)$$

$$\supset \left(/value1 = V \ \wedge \ /stack = S) \supset \nabla^{\langle x_1, x_2 \rangle}_{\langle /stack, /value1 \rangle}(/stack\,/head = V \ \wedge \ /stack\,/tail = S)\right)$$

Performing the ∇ substitutions we get

$$\left(/value1 = V \ \wedge \ /stack = S \supset \uparrow(x_1, /head) = V \ \wedge \ \uparrow(x_1, /tail) = S\right) \supset$$

$$\left(/value1 = V \ \wedge \ /stack = S \supset \uparrow(x_1, /head) = V \ \wedge \ \uparrow(x_1, /tail) = S\right)$$

Which is immediate. Therefore, since all of the conditions of the assignment rule are satisfied, the following formula is true for all states

$$\{nonprefix(\langle /stack, /value1 \rangle) \wedge /value1 = V \ \wedge \ /stack = S\}$$

$$push(\langle /stack, /value1 \rangle)$$

$$\{/stack\,/head = V \ \wedge \ /stack\,/tail = S\}$$

4.5. Chapter Summary

We have now developed a system of formal proof for Hg programs satisfying the restriction that all procedure calls have actual parameters satisfying the nonprefix property in the calling state. By way of example the reader is shown how to prove program segments to be correct, and it is clear that the assignment axiom produces quite complex preconditions even for simple statements. Fortunately we are able to simplify such proofs by applying a theorem which says that a selector selecting a node in the root graph of the state can never be aliased. Although this theorem can remove from consideration conjunctions involving terms of the form $eq(s,g)$ where $s \in Sel$ and $g \in Gsel$, it can do nothing to remove terms of the form $eq(s_1,s_2)$ where neither s_1 nor s_2 selects a node in the root graph of the state. In Chapter 5, we introduce a type system into Hg and prove a theorem which permits us to remove such terms under certain conditions.

Chapter 5

Introducing Types into Hg Programs and Proofs

The definition of the language Hg and the verification system developed in the preceding chapters ignores the question of data types completely. In this chapter we extend the notion of an h–graph to that of a typed h–graph and then develop a method for including type information in Hg programs.

Researchers have different views on what constitutes a data type. Some insist that data types are best represented by algebras [34, 35], while others take a more traditional approach and treat data objects as a set of values together with some operations on those values [36]. In this work, we characterize the type of an object (a node) by giving the set of all possible values that object may take on. The use of data types in programs can be thought of as describing the valid values each object in a program may take. The problem of *type checking* is that of determining if in any particular program an object may take on a value that is not in its set of valid types.

In the language Hg, a program's data states are described with h–graphs and the sets of values a node may take are described using an h–graph grammar. A program's grammar generates all the valid states of data for the program.

5.1. H–graph Grammars and Typed H–graphs

The term *grammar* is usually applied to a mathematical specification of a set of strings over some alphabet. We define here a method of giving mathematical specifications of a set of h–graphs. We call such a specification an *h–graph grammar*.

Definition 5–1: (h–graph grammar)
Recall that Ξ is our base set of characters. An *h–graph grammar* is a quadruple

$\mathcal{G}=\langle T,\xi,S,P\rangle$ where
T is a finite alphabet, the *typenames*, composed of two disjoint subsets, T_B and T_V, the *BNF expander* and *value* productions, respectively.
$\xi \subset \Xi$ is a finite alphabet, the *character set*, with ξ and T disjoint sets.
$S \in T$ is the *root type*.
P is a finite set of productions, each of the form $lhs ::= rhs$, with $lhs \in T$ such that
if $lhs \in T_B$ then $rhs \in (T_B \cup \xi)^*$, a *BNF expander production*.
if $lhs \in T_V$ then rhs is either
(1) an element of $(T_B \cup \xi)^*$, a *string value production*, or
(2) a quadruple $\langle M,E,s,\mu\rangle$, a *graph value production*, where
$\langle M,E,s\rangle \in \Omega$ is a graph, and
$\mu: M \rightarrow 2^{T_V}$, gives the types of each node in the graph $\langle M,E,s\rangle$.
if $lhs \equiv S$ then rhs must be a graph value production.
Note that string value and BNF expander productions are ordinary BNF (context free) grammar productions.

In a fashion analogous with that of string grammars, we define the h–graph equivalents for sentential forms, direct derivations, derivations, and the language defined by a grammar.

Definition 5–2: (h–graph sentential form)
If $\mathcal{G}=\langle T,\xi,S,P\rangle$ is an h–graph grammar, then an *h–graph sentential form*, h, for $\mathcal{G}$ is a quadruple $\langle G,V,r,\tau\rangle$ where
$G \subset \Omega$, G, finite, is the *graphset*,
V :*nodeset*$(G) \rightarrow G \cup \xi^* \cup 2^{T_V}$, V is the *value function*,
$r \in G$, r is the *root graph*.
τ:*nodeset*$(G) \rightarrow T$ is the *node type function*, τ is partial and defined only for $n \in$ *nodeset*(G) where $V(n) \in G \cup \xi^*$.
The *initial h–graph sentential form* for $\mathcal{G}$ is the quadruple $\langle \emptyset,\emptyset,S,\emptyset\rangle$.

A sentential form in a string grammar is a string composed of a mix of terminal and nonterminal symbols. An h–graph sentential form is similar in that some nodes have terminal (graph or atom) values and other nodes have non–terminal (typename set) values. In addition, each terminal node is tagged with the typename used to derive its value.

Definition 5–3: (direct derivation in an h–graph grammar)
Given the grammar $\mathcal{G}=\langle T,\xi,S,P\rangle$, and an h–graph sentential form h':

(1) $h'=\langle G',V',r',\tau'\rangle$ is *directly derived* from the initial h–graph sentential form $h=\langle\emptyset,\emptyset,S,\emptyset\rangle$, by application of the production $S::=\langle M,E,s,\mu\rangle$, written $h\underset{\mathcal{G}}{\Rightarrow}h'$, if h' is the result of the following steps:

let $\bar{m}$ be a tuple of the nodes in M, let $\bar{k}$ be a tuple of nodes, $|\bar{k}|=|M|$, and let $g^*=\langle M,E,s\rangle^{\bar{k}}_{\bar{m}}$. Define $G'=\{g^*\}$, $V'=\mu^{\bar{k}}_{\bar{m}}$, $r'=g^*$, and $\tau'=\emptyset$.

(2) $h'=\langle G',V',r,\tau'\rangle$ is *directly derived* from the h–graph sentential form $h=\langle G,V,r,\tau\rangle$, by application of the production $lhs::=rhs\in P$, written $h\underset{\mathcal{G}}{\Rightarrow}h'$, if $\exists n\in nodeset(G)$ such that $V(n)\in 2^{T_V}$ and $lhs\in V(n)$, and h' is the result of the following steps applied to h:

(a) if rhs is a string value production, then choose some atom a derived from rhs in the usual fashion. We assume the reader is familiar with the notion of a context free string grammar and derivation thereof, see for example [37]. Define $G'=G$, $V'=V\{a:n\}$, and $\tau'=\tau\hat{}\langle n,lhs\rangle$.

(b) if rhs is a graph value production, $rhs=\langle M,E,s,\mu\rangle$, then perform one of the following operations:

(i) let $\bar{m}$ be a tuple of the nodes in M and let $\bar{k}$ be a tuple of nodes disjoint from $nodeset(G)$, $|\bar{k}|=|M|$, and let $g^*=\langle M,E,s\rangle^{\bar{k}}_{\bar{m}}$. Define $G'=G\hat{}g^*$, $V'=V\{g^*:n\}\cup\mu^{\bar{k}}_{\bar{m}}$, and $\tau'=\tau\hat{}\langle n,lhs\rangle$, or

(ii) for some node $m\in nodeset(G)$ such that $\tau(m)=lhs$, define $G'=G$, $V'=V\{V(m):n\}$, and $\tau'=\tau\hat{}\langle n,lhs\rangle$.

A direct derivation in a string grammar replaces a nonterminal α in a sentential form by one of the right hand sides of a production with left hand side α. Similarly, a direct derivation in an h–graph grammar replaces a node's type value with an element of that type. In the case that the type is replaced by a graph value, this can be accomplished in one of two ways, as reflected above by alternatives (b.i) and (b.ii). Either a new graph of the given type is constructed or a graph of the given type which already appears as the value of some node is used. This second alternative permits a derivation to produce an h–graph in which aliasing occurs. We call a derivation which does not apply rule (b.ii), a derivation *without copying*. The following theorem shows that aliases are only introduced by copying provided the grammar productions themselves do not contain right hand side graphs with multiple paths to the same node.

Theorem 5–1: Let h be an h–graph sentential form for grammar $\mathcal{G}$, and $h\underset{\mathcal{G}}{\Rightarrow}h'$ *without copying* using the graph value production $lhs::=\langle M,E,s,\mu\rangle$. If there are

no aliases in h and no aliases in $g = \langle M,E,s \rangle$, then there are no aliases in h'.

Proof: Straightforward by induction on the complexity of h.

□

In writing expressions involving derivations, $\Rightarrow$ is written for $\underset{\mathcal{G}}{\Rightarrow}$, when the grammar $\mathcal{G}$ is clear from the context.

Definition 5–4: (derivation in an h–graph grammar)
Let $\mathcal{G}=\langle T,\xi,S,P \rangle$ be an h–graph grammar and h an h–graph sentential form for $\mathcal{G}$. We say that an h–graph sentential form h' is *derived* from h using $\mathcal{G}$, written $h \overset{*}{\Rightarrow} h'$, iff

$$\exists h_0,h_1,\cdots,h_n,\ h=h_0 \wedge h_n=h' \wedge \forall i,0 \leqslant i < n, h_i \Rightarrow h_{i+1}$$

using a production from P.

Definition 5–5: (language of an h–graph grammar)
The *language defined by* $\mathcal{G}=\langle T,\xi,S,P \rangle$ is the set of h–graph sentential forms $\mathcal{L}_{\mathcal{G}}$, such that $h=\langle G,V,r,\tau \rangle \in \mathcal{L}_{\mathcal{G}}$ *iff* ,

(1) $h \neq \langle \emptyset,\emptyset,S,\emptyset \rangle$
(2) $\forall n \in nodeset(G),\ V(n) \in G \cup \xi^*$
(3) $\langle \emptyset,\emptyset,S,\emptyset \rangle \overset{*}{\Rightarrow} h$

Definition 5–6: (type correct h–graph)
A *typed h–graph* $h=\langle G,V,r,\tau \rangle$ in the context of a grammar $\mathcal{G}=\langle T,\xi,S,P \rangle$ is an h–graph $\langle G,V,r \rangle$ together with the function τ that associates a typename in $\mathcal{G}$ with each node n in $nodeset(h)$. The typename $\tau(n)$ designates the type of value node n contains in h. We term this typename $\tau(n)$ *the type of node n in h.* If there is a derivation $S \Rightarrow r$, and for each node in $nodeset(h)$ one of the following holds:

(1) $V(n)$ is an atom and there is a BNF derivation $\tau(n) \overset{*}{\Rightarrow} V(n)$ in $\mathcal{G}$ or
(2) $V(n)$ is a graph $\langle M_n,E_n,s_n \rangle$ which is a copy of graph $\langle M,E,s \rangle$ and there is a production $\tau(n)::=\langle M,E,s,\mu \rangle$ and furthermore for each node $m \in M_n$ $\tau(m) \in (m')$, where m' is the node in M corresponding to m in M_n

then h is said to be *type correct with respect to* $\mathcal{G}$.

We will say that an h–graph is *type correct* to indicate type correctness with respect to a grammar known in context.

Type correctness is a desirable property for h–graphs. In a type correct h–graph h, if $\tau(n) = t$, then the value of n is an element of the language defined by the typename t, that is, it is a value of type t.

Theorem 5–2: If $h \in \mathcal{L}_{\mathcal{G}}$, then h is type correct with respect to $\mathcal{G}$.

Proof: Immediate from the definition of a derivation (Definition 5–4).

□

To incorporate the idea of data type into Hg programs, we require that the set of states *Stat* consist exclusively of typed h–graphs. Henceforth whenever we use the term *h–graph* we are referring to a typed h–graph. The relevant grammar $\mathcal{G}$ may be inferred from the context in general.

Given an h–graph grammar $\mathcal{G}$, and an arbitrary selector s, it makes sense to associate a set of types with s, designating the various types of nodes that s can select when applied to the h–graphs in $\mathcal{L}_{\mathcal{G}}$. This typeset can be determined by inspecting the h–graphs in $\mathcal{L}_{\mathcal{G}}$. Of equal interest, though, is a function which can tell us for a selector s in an h–graph h what types of values s might have in h and what types of values could be assigned to the node selected by s and result in an h–graph h' such that $h' \in \mathcal{L}_{\mathcal{G}}$. We introduce the function $\theta_{\mathcal{G}}$ to tell us what this set of potential assignment types is for any given selector.

Definition 5–7: ($\theta_{\mathcal{G}}$, selector typeset function)
$\theta_{\mathcal{G}}$:*Sel* $\rightarrow 2^{T_V}$ is the *selector typeset function* for the grammar $\mathcal{G}$.
$\theta_{\mathcal{G}}(s)$ for the grammar $\mathcal{G}=\langle T,\xi,S,P\rangle$ is defined recursively as follows:
(1) if $s \equiv gsel$ is a simple selector, then let P^* be the set of all productions of the form $S::=rhs$ in P where $rhs = \langle M_p,E_p,s_p,\mu_p\rangle$ and $[\![gsel]\!]\langle M_p,E_p,s_p\rangle \neq \perp$.
If $\forall p,q \in P^*$, $\mu_p([\![gsel]\!]\langle M_p,E_p,s_p\rangle) = \mu_q([\![gsel]\!]\langle M_q,E_q,s_q\rangle)$, then

$$\theta_{\mathcal{G}}(s)=\mu_p([\![gsel]\!]\langle M_p,E_p,s_p\rangle), \text{ for any } p \in P^*,$$

otherwise

$$\theta_{\mathcal{G}}(s)=\varnothing$$

(2) if $s \equiv gsel_1 \cdots gsel_n$ is a composite selector, then let P^* be the set of all graph value productions $p = lhs ::= rhs$ such that $lhs \in \theta_{\mathcal{G}}(gsel_1 \cdots gsel_{n-1})$ and $rhs = \langle M_p, E_p, s_p, \mu_p \rangle$ and $[\![gsel]\!] \langle M_p, E_p, s_p \rangle \neq \perp$.
If $\forall p,q \in P^*$, $\mu_p([\![gsel_n]\!] \langle M_p, E_p, s_p \rangle) = \mu_q([\![gsel_n]\!] \langle M_q, E_q, s_q \rangle)$, then

$$\theta_{\mathcal{G}}(s)=\mu_p([\![gsel_n]\!] \langle M_p, E_p, s_p \rangle), \text{ for any } p \in P^*,$$

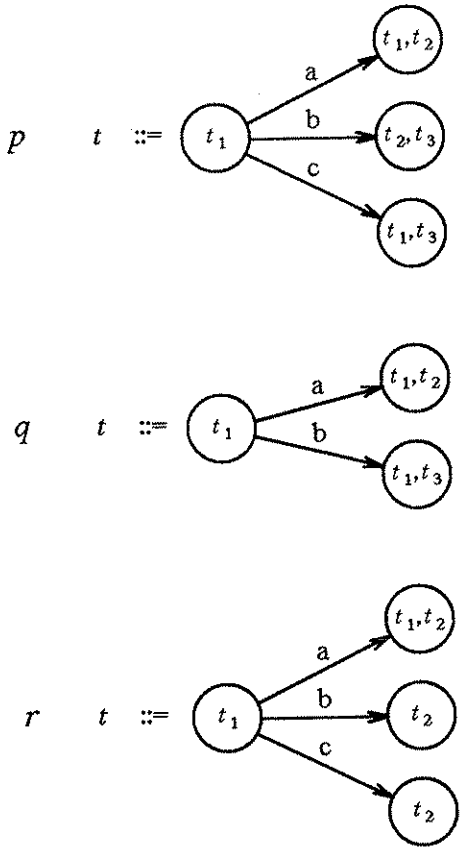
otherwise

$$\theta_{\mathcal{G}}(s)=\varnothing$$

The definition of θ requires that if a particular selector appears in several h–graph grammar productions then that selector must have identical typesets in each of those productions, otherwise its typeset is the emptyset. If $\theta_{\mathcal{G}}(s)=\varnothing$, this does not necessarily mean that there are no values which could be assigned to s and result in an h–graph in $\mathcal{L}_{\mathcal{G}}$, but only that due to the occurrence of different typesets for the selector s we cannot determine in advance exactly which value types assigned to s would result in an h–graph in $\mathcal{L}_{\mathcal{G}}$. If we attempted to take the intersection of the types which s can select, then $\theta_{\mathcal{G}}(s)$ would not describe some of the valid h–graphs in $\mathcal{L}_{\mathcal{G}}$. The empty set may also be returned by $\theta_{\mathcal{G}}(s)$ if s is not a selector defined in any of the h–graphs in $\mathcal{L}_{\mathcal{G}}$. Figure 5.1 shows an example of how to determine $\theta_{\mathcal{G}}$ for several selectors.

When the context is clear, we use θ instead of $\theta_{\mathcal{G}}$ to refer to the selector typeset function of the grammar $\mathcal{G}$. We do this in particular when θ is the selector typeset function of the grammar for some particular program or procedure.

Suppose in some grammar, the productions represented below are all those with left hand side t. Each production is represented by a production identifier, the left hand side typename, and a right hand side graph. The typeset given to a right hand side node by the type function μ of that production is written inside the circle representing that node. We use the notation $graph(p)$ to denote the *rhs* graph of the production p.



Supposing $\theta(s)=\{t\}$. Then

$\mu_p([\![\,/\,]\!](graph(p))) = \mu_q([\![\,/\,]\!](graph(q))) = \mu_r([\![\,/\,]\!](graph(r))) = \{t_1\}$, so $\theta(s/)=\{t_1\}$

$\mu_p([\![\,/a\,]\!](graph(p))) = \mu_q([\![\,/a\,]\!](graph(q))) = \mu_r([\![\,/a\,]\!](graph(r))) = \{t_1,t_2\}$, so $\theta(s/a)=\{t_1,t_2\}$

$\mu_p([\![\,/b\,]\!](graph(p))) \neq \mu_q([\![\,/b\,]\!](graph(q))) \neq \mu_r([\![\,/b\,]\!](graph(r)))$, so $\theta(s/b)=\emptyset$

$\mu_p([\![\,/c\,]\!](graph(p))) \neq \mu_r([\![\,/c\,]\!](graph(r)))$, so $\theta(s/c)=\emptyset$

Figure 5.1 Example showing computation of θ.

Theorem 5–3:
Let $\mathcal{G}$ be an h–graph grammar, $\sigma=\langle G,V,r,\tau\rangle$ an h–graph type correct with respect to $\mathcal{G}$, then for all $s \in Sel$, either
(1) $\tau_\sigma([\![s]\!](\sigma)) \in \theta_{\mathcal{G}}(s)$, or
(2) $[\![s]\!](\sigma) = \perp$

Proof: The proof is straightforward by induction on the number of graph selectors in s. If s consists of a single graph selectors, the result is immediate from comparison of part (1) of the definition of $\theta_{\mathcal{G}}$ (Definition 5–7) and the definition of a derivation in an h–graph grammar (Definition 5–4).

□

We now extend the notion of a program to include an h–graph grammar. First, we incorporate a grammar into procedure definitions. As shown below, this grammar is used to describe all the valid states that a procedure may assume.

Definition 5–8: (*Pdef* , procedure definitions extended)
Pdef is the domain of procedure definitions. Each definition, $d \in Pdef$ is a quadruple $\langle \sigma,K,\bar{\eta},\mathcal{G}\rangle$, where
$\sigma \in \mathcal{L}_{\mathcal{G}}$ is an h–graph, describing the *initial state* of the procedure;
K is the *code* for the procedure;
$\bar{\eta} \in Sel^{N}$ is a tuple of selectors, the *formal parameters* of the procedure, such that $\forall i, 1 \leq i \leq |\bar{\eta}|, [\![\bar{\eta}\downarrow i]\!](\sigma) \in nodeset(r)$, and
$\forall j,1 \leq j \leq |\bar{\eta}|, i \neq j, [\![\bar{\eta}\downarrow i]\!](\sigma) \neq [\![\bar{\eta}\downarrow j]\!](\sigma)$
and $\mathcal{G}=\langle T,\xi,S,P\rangle$ is an h–graph grammar, the *local state grammar*, such that $\langle \emptyset,\emptyset,S,\emptyset\rangle \stackrel{*}{\Rightarrow} \sigma$ without copying.

Since a procedure shares argument values with its calling scope, we need to make provisions for the types of arguments to match in the procedure's state grammar and the calling scope's state grammar. To support this agreement between procedures and their callers we introduce co–grammars and then incorporate this idea into the definition of Hg programs. The reorganization of the function *Comp*, which defines the semantics of Hg programs, allows use of this information in determining which procedure calls are well defined and which are not.

Definition 5–9: (co–grammar of an h–graph grammar)

A *co-grammar* $\mathcal{G}_c=\langle T_c,\xi_c,S_c,P_c\rangle$ of an h-graph grammar $\mathcal{G}=\langle T,\xi,S,P\rangle$ is an h-graph grammar such that for all $t\in T_c\cap T$, P and P_c agree exactly on all productions of the form $t::=rhs$. Such typenames t are called *consensus types*.

Definition 5-10: (*Prog*, Hg programs)
The class of programs, *Prog*, consists of quadruples of the form $\langle\sigma,K,\pi,\mathcal{G}\rangle$.
$\sigma\in\mathcal{L}_{\mathcal{G}}$ is the *initial state* of the program,
$K\in Code$ is the *code* for the program,
π defines the set of *procedure definitions* for the program, and
$\mathcal{G}=\langle T,\xi,S,P\rangle$ is an h-graph grammar such that $\langle\emptyset,\emptyset,S,\emptyset\rangle\stackrel{*}{\Rightarrow}\sigma$ without copying. $\mathcal{G}$ is the *local state grammar* of the program.
We also require that $\forall p,q\in range(\pi)$ with local state grammars $\mathcal{G}_p$ and $\mathcal{G}_q$ respectively, $\mathcal{G}_p$, $\mathcal{G}_q$ are co-grammars and each is a co-grammar of $\mathcal{G}$.

We require in the above definition, that the initial states of programs and their procedures must be derived without copying from an initial sentential form. This restriction is required so no difficult aliasing patterns will appear in the initial state. Such restrictions are typical of most programming languages. In Pascal, for example, one cannot create a program in which two local variables are aliased with each other.

An h-graph representing the state of computation of an Hg program will contain information on the actual types of the nodes in that state. In addition, from the grammar we can determine the *valid* types of any selector, that is the types that the state grammar permits that selector to take on.

We want to use the type information provided by our h-graph grammars to assure that in any Hg program, the assignments and condition evaluations that take place are type correct, in the usual sense of the word. We are concerned with name strict compatibility of types, that is, only objects whose types have the same name are compatible.

Since assignment of an expression value to a selector can affect the type structure of the state we must assure that expressions are used in a manner that preserves the correct type structure of a program. To do this we need to know the types of arguments and results of the primitive functions of Hg. It is not necessary to consider boolean

expressions and predicates since they cannot appear in a context which causes direct modification of the type structure of an Hg program state. The next few definitions provide this type information.

Definition 5–11: (arity of functions, α)
α:*Func* $\rightarrow$**N+**, maps a particular function symbol into the arity of the function it represents, i.e., the number of arguments the function takes.

We assume there is a predefined set $\hat{T}$ of base types over which the primitive functions in *Func* are defined. The next definition provides the basic information about valid argument and result types for these primitives.

Definition 5–12: (types of function calls, ϕ)
The function ϕ maps its arguments, a function symbol, f, and a tuple from $\hat{T}^{\mathbf{N}}$, of length $\alpha(f)$, into a single base type. Formally, we have ϕ:*Func* $\times \hat{T}^{\mathbf{N}} \rightarrow \hat{T} \hat{} \perp$.

The definition of each function symbol of the underlying logical structure $\mathcal{U}$ is assumed to match the definition of argument and result types specified by ϕ. Note that *overloading* of function symbols is allowed, i.e., a function may return a result of one of several types depending on the types of the arguments it receives; however the arity of a function is fixed.

Since any given selector may take on several types in the course of a program's execution, it may be impossible to statically determine what the type of an expression in a program may be. There is, however, some information available through the use of the typeset function θ. We can extend θ of any program grammar to operate on expressions from the class *Expr* in the following manner:

Definition 5–13: (potential expression type, θ^*)
Let $\mathcal{P}$ be a program or procedure with local state grammar $\mathcal{G}$, T be the typeset of $\mathcal{G}$, and e an expression in the code part of $\mathcal{P}$. θ^*:*Expr* $\rightarrow (2^{T \hat{} \perp})^{\mathbf{N}}$ is defined as follows:

if $e \equiv s$, then $\theta^*(e)=\theta(s)$

if $e \equiv f(e_1,\cdots,e_n)$, then $\theta^*(e)=\bigcup_{t_i \in \theta^*(e_i)} \phi(f,\langle t_1,\cdots t_n \rangle)$

Although the type of an expression may not be statically determinable, at any point in a program's execution the value of an expression will have only one type. We can extend the function τ to give us the actual type of the value of an expression in any state.

Definition 5–14: (actual type of an expression in a given state, τ_σ^*)
Let $\sigma = \langle G, V, r, \tau \rangle$ be a state of a program with typeset T.
We define $\tau_\sigma^*: Expr \rightarrow T \hat{} \perp$ as follows:
if $e \equiv s$, then $\tau_\sigma^*(e) = \tau([\![s]\!](\sigma))$
if $e \equiv f(e_1, \cdots, e_n)$, then $\tau_\sigma^*(e) = \phi(f, \langle \tau_\sigma^*(e_1), \cdots, \tau_\sigma^*(e_n) \rangle)$

Now that we understand what it means to say that an expression has a particular type, let us extend the definition of Hg to follow our intuition about types.

One aspect of types we would like to insure is that when a node selected by a selector is assigned a value, the selector must have the value's type as one of its valid types in order for the assignment to work. And the actual type of the node selected by the selector then becomes the type of the assigned value. This means we must modify the assignment alternative of the definition of *Comp*. To do so, we make the following definition:

Definition 5–15: ($\sigma\{v, t:s\}$, typed variant of state)
The notation $\sigma\{v, t:s\}$ where $\sigma = \langle G, V, r, \tau \rangle \in Stat$, $v \in G \cup \Delta \hat{} \perp$, and $t \in T \hat{} \perp$, denotes the state, $\sigma' = \langle G, V\{v:[\![s]\!](\sigma)\}, r, \tau\{t:[\![s]\!](\sigma)\} \rangle$ if $[\![s]\!](\sigma) \neq \perp$, and $\perp$ if $[\![s]\!](\sigma) = \perp$.

This signifies the state σ' resulting from a substitution of v for the value of the node n selected by s in σ in the value function, V, of σ, and t for the type of the node n in the type function, τ, of σ.

We define the extension of a typed state variant to vectors in the usual fashion: Given $\sigma = \langle G, V, r, \tau \rangle$, and $|\bar{v}| = |\bar{t}| = |\bar{s}| = m$, and

$$\forall i, 1 \leq i \leq m, \bar{v}\downarrow i \in (G \cup \Delta \hat{} \perp), \bar{t}\downarrow i \in T \hat{} \perp, \bar{s}\downarrow i \in Sel$$

then $\sigma\{\bar{v}, \bar{t}:\bar{s}\}$ is defined as follows:

$$\sigma\{\bar{v},\bar{t}:\bar{s}\} = \begin{cases} \sigma\{\bar{v}\downarrow 1,\bar{t}\downarrow 1:\bar{s}\downarrow 1\}\{\langle \bar{v}\downarrow 2,\cdots,\bar{v}\downarrow m\rangle,\langle \bar{t}\downarrow 2,\cdots,\bar{t}\downarrow m\rangle:\langle \bar{s}\downarrow 2,\cdots,\bar{s}\downarrow m\rangle\} & \text{if } m>1 \\ \sigma\{\bar{v}\downarrow 1,\bar{t}\downarrow 1:\bar{s}\downarrow 1\} & \text{if } m=1 \end{cases}$$

The state sequence function *Comp* can now be modified to behave in a way that corresponds with our intuition about types. In particular we change the way Comp behaves on assignment statements and procedure calls. In an assignment of the value of expression e to selector s, the type of the value of e at the time of assignment will be returned as the type of the node selected by s in the resulting state. In a procedure call, since the values of the actual parameters are assigned to the formals on procedure entry and the values of the formals are assigned to the actuals on exit, it is natural to carry the type information into and out of the procedure. We add this requirement to those already set forth for procedure calls.

Definition 5–16: (*Comp* $_P$ with typed h–graphs)
Comp $_P$ is as defined in Definition 2–30, with the following exceptions:
Wherever an h–graph appears in *Comp* $_P$ it is replaced by a corresponding typed h–graph. Let $\mathcal{G}$ be the local state grammar of the program or procedure in which the assignment or procedure call appears.
The assignment alternative is modified as follows:
$s := e \rightarrow \sigma\{R(e)(\sigma),\tau_\sigma{}^*(e):s\}$

In other words, if the selector on the left hand side of an assignment is defined in the starting state, then in the resulting state, the node it selects will contain the value of the expression e and the node will have the type of the value of e, otherwise the resulting state is $\perp$.
The procedure call alternative must be modified as follows:

$$p(\bar{\alpha}) \rightarrow \begin{cases} \langle \sigma',\sigma''\rangle \frown \mathit{Comp}_P(K_p)(\sigma'') \frown \langle \sigma'''\rangle \\ \qquad \text{if } \pi(p)=\langle \sigma_p,K_p,\bar{\eta}_p,\mathcal{G}_p\rangle \text{ and } |\bar{\eta}_p| = |\bar{\alpha}| \\ \perp \text{ otherwise} \end{cases}$$

σ', σ'', and σ''' are defined below, and $\bar{\alpha} \in Sel^N$.
The first alternative applies if p is the name of a procedure defined in this state and the number of the parameters of the call and the procedure definition are identical.
$\sigma' = \langle G',V',r',\tau'\rangle$ is constructed as follows:

Given that $\sigma_p = \langle G_p,V_p,r_p,\tau_p\rangle$, and $\sigma=\langle G,V,r,\tau\rangle$,
1) form a tuple $\bar{n}$ from all the nodes in $nodeset(\sigma_p)$.
2) Choose a set of $|\bar{n}|$ nodes from $\Phi \sim (nodeset(\sigma) \cup nodeset(\sigma_p))$ and form a

tuple $\bar{m}$ from these.

$G' = G \cup G_{p\frac{\bar{m}}{\bar{n}}}$, $V' = V \cup V_{p\frac{\bar{m}}{\bar{n}}}$, $r' = r_{p\frac{\bar{m}}{\bar{n}}}$, and $\tau' = \tau \cup \tau_{p\frac{\bar{m}}{\bar{n}}}$

$\sigma'' = \sigma'\{R(\bar{\alpha})(\sigma), \tau_\sigma{}^*(\bar{\alpha}):\bar{\eta}_p\}$

$\sigma''' = \omega\{R(\bar{\eta}_p)(Last(Comp\,[\![K_p]\!](\sigma''))), \tau_\omega{}^*(\bar{\eta}):\bar{\alpha}\}$

Where $\omega = \langle G_\omega, V_\omega, r_\omega, \tau_\omega \rangle$, and $\langle G_\omega, V_\omega, r_\omega, \tau_\omega \rangle = Last(Comp\,[\![K_p]\!](\sigma''))$. That is, ω is a state identical to $Last(Comp\,[\![K_p]\!](\sigma''))$, but with the same root-graph as the calling state.

5.2. Exploiting the Typed Model

Merely equipping programs with grammars and extending *Comp* to make use of the information these grammars provide does not guarantee that programs will now be correct and no aliasing will occur. Such information can, however, provide us with extra assurances about program correctness and help simplify the task of verifying programs in which aliases can occur.

Before we can use type information in correctness arguments we must overcome a deficiency in our view of defining a program's meaning with a state sequence. *Comp* defines Hg by providing a function from states into state sequences. The sequence generated by *Comp*, though, is too detailed for discussion of some program properties. If, for example, we want to describe invariant properties of a main program selector, it is clear we don't want to discuss these properties in states which are introduced in a procedure call, for in those states main program selectors may not be defined, and even if they are they most likely do not select the same node as in the main program state. We only want to discuss such properties in states which are at the top level of the calling chain. In addition, any inaccessible graphs which result from a procedure call are of no interest to us. Hence we first present the definition of the set of selectable nodes of an h-graph and a function which trims away the non-selectable parts of a state h-graph, then a function providing the top level state sequence of a program

Definition 5-17: (*selectable*(h), the set of selectable nodes of h-graph h)
Given $h = \langle G, V, r \rangle$, $r = \langle M, E, s \rangle$, the set of nodes *selectable* in h, written

selectable(h) is defined recursively as follow:
(i) $M \subseteq$ *selectable*(h)
(ii) If $n \in$ *selectable*(h) and $V(n)=\langle M_n, E_n, s_n \rangle$, then $M_n \subseteq$ *selectable*(h).
(iii) Nothing is in *selectable*(h) unless it follows from (i), and (ii).

Definition 5–18: (*trim*(h), h–graph h with inaccessible parts removed)
Given $h=\langle G, V, r, \tau \rangle$ The h–graph h with inaccessible parts removed, *trim*(h) $= \langle G', V', r, \tau' \rangle$ where
G' is defined by the following two rules:
(1) $r \in G'$,
(2) if $n \in$ *selectable*(h) and $V(n) \in G$, then $V(n) \in G'$.
$V' = V|$*selectable*(h), and
$\tau' = \tau|$*selectable*(h).

Definition 5–19: ($Tcomp_{\mathcal{P}}$, top level computation sequence for program *Prog*)
$Tcomp_{\mathcal{P}}$: (*Code*) $\rightarrow$ (*Stat* $\rightarrow$ *Stat*$^{\omega}$) is a function mapping statement sequences in a program into functions from states to state sequences similar to *Comp*. The primary difference between *Tcomp* and *Comp* is that a state sequence function generated for K by *Tcomp* does not include the state sequence of any procedure called in K. The state sequence function defined by application of $Tcomp_{\mathcal{P}}$ to a program $\mathcal{P}=\langle \sigma, K, \pi, \mathcal{G} \rangle$ is defined below for each case of *Code*. Alternatives for *Code* appear with the corresponding value for $Tcomp_{\mathcal{P}}(Code)(\sigma)$ to the right of the arrow, "$\rightarrow$".

Cases of *Code*
$\epsilon \rightarrow \langle \sigma \rangle$
$s := e \rightarrow Comp_{\mathcal{P}}(s := e)(\sigma)$
$k;K \rightarrow Tcomp_{\mathcal{P}}(k)(K)^{\frown} Tcomp_{\mathcal{P}}(K)(\sigma)$
if b **then** K_1 **else** K_2 **endif** $\rightarrow$

$$\begin{cases} Tcomp_{\mathcal{P}}(K_1)(\sigma) & \text{if } B(b)(\sigma) = \mathbf{T} \\ Tcomp_{\mathcal{P}}(K_2)(\sigma) & \text{if } B(b)(\sigma) = \mathbf{F} \\ \langle \perp \rangle & \text{otherwise} \end{cases}$$

while b **loop** K **endloop** $\rightarrow$

$$\begin{cases} \langle \sigma \rangle^{\frown} Tcomp_{\mathcal{P}}(K)(\sigma)^{\frown} Tcomp_{\mathcal{P}}(\textbf{while } b \textbf{ loop } K \textbf{ endloop})(Last(Tcomp_{\mathcal{P}}(K)(\sigma))) \\ \qquad \text{if } B(b)(\sigma) = \mathbf{T} \\ \langle \sigma \rangle \text{ if } B(b)(\sigma) = \mathbf{F} \\ \langle \perp \rangle \text{ otherwise} \end{cases}$$

$p(\overline{\alpha}) \rightarrow trim(Last(Comp(p(\overline{\alpha}))(\sigma)))$

The following theorem is quite useful in removing arguments involving *eq* from assertions. Informally stated, the theorem says that if in the initial state of a program there are no aliases of a selector s selecting a node in the rootgraph of the initial state, then no aliases will be introduced in execution of the program.

Theorem 5–4: Given $\mathcal{P} = \langle \sigma, K, \pi, \mathcal{G} \rangle$, $\sigma = \langle G, V, r, \tau \rangle$ such that
$\forall s$, if $[\![s]\!](\sigma) = n$, $n \in nodeset(r)$ then $\forall u \in Sel$, $u \neq s$, $[\![\neg eq(u, s)]\!](\sigma)$.
Let σ_i represent $Tcomp(K)(\sigma)\downarrow i$, $1 \leq i \leq |Tcomp(K)(\sigma)|$
Then

$$\forall u \in Sel - \{s\}, [\![\neg eq(u, s)]\!](\sigma_i)$$

Given some procedure name $p \in pnames(\mathcal{P})$, $\pi(p) = \langle \sigma_p, K_p, \bar{\eta}_p, \mathcal{G}_p \rangle$,
$\sigma_p = \langle G_p, V_p, r_p, \tau_p \rangle$, such that $\forall s$, if $[\![s]\!](\sigma_p) = n, n \in nodeset(r_p)$,
then $\forall u \in Sel$, $u \neq s$, $[\![\neg eq(u, s)]\!](\sigma_p)$. Let σ_j represent
$Tcomp(K_p)(\sigma_p)\downarrow j$, $1 \leq j \leq |Tcomp(K_p)(\sigma_p)|$
Then

$$\forall u \in Sel, u \neq s, [\![\neg eq(u, s)]\!](\sigma_j)$$

Proof: We must look at the cases of *Tcomp* to determine the validity of this theorem. Since the states σ and σ_p are each well formed h–graphs, satisfying the condition that there are no aliases of any selectors selecting nodes in the root–graph, we can apply Theorem 3–8 to get our result for assignment statements in *Tcomp*, Theorem 4–14 to get our result for procedure calls in *Tcomp*, and use induction for the other cases of *Tcomp*.

□

Many programming languages support the notion of data types. Compilers written for these languages often perform some kind of *type checking* to determine if typed elements of programs manipulate types in a manner consistent with the semantics of the language. Such tests are usually easy to perform and are quite useful to programmers in pointing out minor program errors. In Section 5.1 we define what it means to say that a state is type correct with respect to a grammar. We now define what we mean by a *type secure* program and show what we can do to assure statically, that is by looking only at the definition of a program, that a program is type secure.

Definition 5–20: (Type secure program)

We say that a program, $\mathcal{P}=\langle\sigma,K,\pi,\mathcal{G}\rangle$, is *type secure* iff for each $\sigma_i \in Tcomp(K)(\sigma)$, σ_i is type correct with respect to $\mathcal{G}$, and for every procedure name $p \in pnames(\mathcal{P})$, $\pi(p)=\langle\sigma_p,K_p,\bar{\eta}_p,\mathcal{G}_p\rangle$ then for all states $\sigma^* \in \mathcal{L}_{\mathcal{G}_p}$, and for each $\sigma_j \in Tcomp(K_p)(\sigma^*)$, σ_j is type correct with respect to $\mathcal{G}_p$.

The next theorem provides a bridge between type correctness and h-graph grammars which is necessary in order to incorporate information derived from a program's grammar into a proof of some assertions about the program.

Theorem 5-5: If σ is type correct with respect to $\mathcal{G}$, then $trim(\sigma) \in \mathcal{L}_{\mathcal{G}}$

Proof: It is apparent from Definition 5-6, that for if $trim(\sigma)$ contains a single graph then there exists a derivation of $trim(\sigma)$ in $\mathcal{G}$. Induction on the complexity of $trim(\sigma)$ gives us the desired result.

□

Now that we have a definition of a type secure program, we would like to develop a method of determining statically that a program is type secure, that is, it is type correct for all states it can reach during its execution. This will guarantee that we can never introduce a type incorrect state into the program sequence through the assignment of an inappropriate value. In defining static type security of a program we need the following concept:

Definition 5-21: (Total selector over a grammar)
A selector s is said to be *total over the grammar* $\mathcal{G}$ if for all $h \in \mathcal{L}_{\mathcal{G}}$, $[\![s]\!](h) \neq \perp$.

Definition 5-22: (Static type security of a program)
We say that a program, $\mathcal{P}=\langle\sigma,K,\pi,\mathcal{G}\rangle$, is *statically type secure* iff it satisfies each of the following requirements:

(1) The initial state σ is type correct with respect to $\mathcal{G}$,

(2) for each assignment, $s := e$, in K
$$\theta_{\mathcal{G}}{}^*(e) \subseteq \theta_{\mathcal{G}}(s),\ \theta_{\mathcal{G}}{}^*(e) \neq \emptyset,$$

(3) for each procedure p in $domain(\pi)$, such that $\pi(p)=\langle\sigma_p,K_p,\bar{\eta}_p,\mathcal{G}_p\rangle$, for each assignment $s := e$, in K_p
$$\theta_{\mathcal{G}_p}{}^*(e) \subseteq \theta_{\mathcal{G}_p}(s),\ \theta_{\mathcal{G}_p}{}^*(e) \neq \emptyset,$$

(4) for each selector $s \in selectors(K)$, s is total over $\mathcal{G}$,

(5) for each procedure call $p(\bar{\alpha})$ in K with $\pi(p)=\langle\sigma_p,K_p,\bar{\eta}_p,\mathcal{G}_p\rangle$,

$$\forall i,\ 1\leqslant i\leqslant|\bar{\alpha}|,\ \theta_{\mathcal{G}}(\bar{\alpha}\downarrow i)=\theta_{\mathcal{G}_p}(\bar{\eta}_p\downarrow i),\text{ and}$$

$$\forall s\in selectors(K_p),\ s\text{ is total over }\mathcal{G}_p,$$

(6) for each procedure q in $domain(\pi)$ and procedure call $p(\bar{\alpha})$ in K_q such that $\pi(p)=\langle\sigma_p,K_p,\bar{\eta}_p,\mathcal{G}_p\rangle$,

$$\forall i,\ 1\leqslant i\leqslant|\bar{\alpha}|,\ \theta_{\mathcal{G}_q}(\bar{\alpha}\downarrow i)=\theta_{\mathcal{G}_p}(\bar{\eta}_p\downarrow i),\text{ and}$$

$$\forall s\in selectors(K_p),\ s\text{ is total over }\mathcal{G}_p,$$

We have defined what we mean when we say a program is statically type secure. We have not however related this notion of static type security to type security of a program. We want to demonstrate the result that any statically type secure program is type secure. Proof of that result requires the following lemma:

Lemma 5–6: Let $\mathcal{G}$ be an h–graph grammar, $s_1,s_2\in Sel$. If $\theta^*(s_1)=\theta^*(s_2)$ then

$$\forall u\in Sel,\ \theta^*(s_2u)\neq\emptyset\Rightarrow\theta^*(s_1u)=\theta^*(s_2u)$$

Proof: By induction

Basis: Suppose u is a simple selector. Since $\theta^*(s_1)=\theta^*(s_2)$ and we know that

$$\forall p,q\in\theta^*(s_2),\text{ and productions }p::=\langle M_p,E_p,s_p,\mu_p\rangle\text{ and }q::=\langle M_q,E_q,s_q,\mu_q\rangle,$$
$$\mu_p([\![u]\!](\langle M_p,E_p,s_p\rangle))=\mu_q([\![u]\!](\langle M_q,E_q,s_q\rangle)).$$

And if $\theta^*(s_2u)\neq\emptyset$, then this set is non empty, so we know that

$$\forall p,q\in\theta^*(s_1),\text{ and productions }p::=\langle M_p,E_p,s_p,\mu_p\rangle\text{ and }q::=\langle M_q,E_q,s_q,\mu_q\rangle,$$
$$\mu_p([\![u]\!](\langle M_p,E_p,s_p\rangle))=\mu_q([\![u]\!](\langle M_q,E_q,s_q\rangle)).$$

And it is clear that

$$\theta^*(s_1u)=\theta^*(s_2u).$$

Induction: Suppose u is a composite selector $gsel_1\cdots gsel_n$. Assume that

$$\theta^*(s_1gsel_1\cdots gsel_{n-1})=\theta^*(s_2gsel_1\cdots gsel_{n-1}).$$

Since $\theta^*(s_2u)\neq\perp$, by the same reasoning as in the base case, we get

$$\theta^*(s_1gsel_1\cdots gsel_{n-1}gsel_n)=\theta^*(s_2gsel_1\cdots gsel_{n-1}gsel_n)$$

And since

$$s_1gsel_1\cdots gsel_{n-1}gsel_n=s_1u$$

and likewise for s_2, we have

$$\theta^*(s_1u)=\theta^*(s_2u)$$

So for all u our result is proved.

□

The next theorem, which is quite powerful, guarantees that when we can make a static determination of a program's type security, we are guaranteed that the program always yields type correct states at run time, and as we saw in Theorem 5–5, such a state is always in the language defined by the grammar of that program, $\mathcal{L}_{\mathcal{G}}$.

Theorem 5–7: If a program is statically type secure, then it is type secure.

Proof: Assume there exists a program $P=\langle\sigma,K,\pi,\mathcal{G}\rangle$ that is statically type secure yet is not type secure. Let the notation σ_l be used as a shorthand for $Tcomp\,\mathcal{K}(K)(\sigma)\downarrow l$.

We will prove the theorem for the procedure call case. The proof for program selectors is a special case of this result.
Let us denote as σ_0 the state σ, the initial state of the procedure call. The proof proceeds by induction. We know that σ_0 is type correct because of the static type security of P, so we can reformulate the predicament of type insecurity as follows: for some l, $0\leq l<|Tcomp\,\mathcal{K}(K)(\sigma)|$ σ_l is type correct but σ_{l+1} is not type correct.
Inspection of the function *Tcomp* reveals that the only two alternatives in which the type information of the h–graph can change are

(1) the assignment statement, and

(2) the procedure call,

so we must show contradictions of our assumption in these two cases and we have proven our theorem.

Proof of case (1)

Let σ_l be a state of $Tcomp\,\mathcal{K}(K)$ in which an assignment statement $s := e$ is executed. Since P is statically type secure, we know that $\theta^*(e)\subseteq\theta^*(s)$. By extension of Theorem 5–3, we can see that $\tau^*(e)\in\theta^*(s)$. So in σ_{l+1}, $\tau([\![s]\!](\sigma))\in\theta^*(s)$, which is a correct type for the node $[\![s]\!](\sigma)$ in $\mathcal{G}$ according to Definition 5–6. Since all other nodes in σ_{l+1} have the same values and types as in σ_l, we see immediately that σ_{l+1} is type correct with respect to $\mathcal{G}$.

Proof of case (2)

Proof of this case proceeds by induction.

Since σ_l is type correct and σ_{l+1} results from the procedure call $p'(\overline{\alpha})$, with $\pi(p') = \langle\sigma_{p'},K_{p'},\overline{\eta}_{p'},\mathcal{G}_{p'}\rangle$, we know that $\sigma_{p'}$ is a type correct state because of the type correctness of the procedure p'. So given a procedure call $p'(\overline{\alpha})$, we know in the sequence generated by *Comp* that σ' (using the notation of Definition 2–30) is type correct. Since static correctness and type correctness of σ_l guarantees

$$\theta(\overline{\alpha}\downarrow i) = \theta_{\mathcal{G}_{p'}}(\overline{\eta}\downarrow i) \text{ for all relevant } i, \text{ we know that } \sigma''$$

is also type correct. By induction, $Last(Comp(K_{p'})(\sigma''))$ is a type correct

state, and once again, since $\theta(\overline{\alpha}\downarrow i) = \theta_{\mathcal{G}_{p'}}(\overline{\eta}\downarrow i)$,

$$Last(Comp(p'(\overline{\alpha}))(\sigma_l)) = Tcomp(p'(\overline{\alpha})(\sigma_l))$$

is a type correct state.

□

We now show how to incorporate information derived from program grammars into formal proofs. We do this by expressing our notions of validity and formal proof with respect to an h–graph grammar.

Definition 5–23: ($\models_{\mathcal{G}} F$, validity of correctness formula with respect to $\mathcal{G}$)
Given $F \in Form$, and an h–graph grammar $\mathcal{G}$, if $[\![F]\!](\sigma) = \mathbf{T}$, for all $\sigma \in \mathcal{L}_{\mathcal{G}}$, then F is said to be *valid with respect to* $\mathcal{G}$, written $\models_{\mathcal{G}} F$.

Definition 5–24: (soundness of inference with respect to a grammar $\mathcal{G}$)
Given an h–graph grammar $\mathcal{G}$, an inference $\dfrac{F_1, \cdots F_n}{F}$ is said to be *sound with respect to* $\mathcal{G}$ if

$$(\forall i, 1 \leqslant i \leqslant n, \models_{\mathcal{G}} F_i) \supset \models_{\mathcal{G}} F$$

Definition 5–25: (formal proof with respect to a grammar)
Given a set of correctness formulae Ax, called the axioms, a set of inference rules Pr, called the proof rules, and an h–graph grammar $\mathcal{G}$, we say that F is formally provable from Ax, Pr, with respect to $\mathcal{G}$,
written

$$\vdash_{Ax, Pr, \mathcal{G}} F$$

whenever there exists $n \geqslant 1$ and a sequence of correctness formulae $F_1, \cdots F_n$ (called the formal proof of F) such that
(i) $F \equiv F_n$
(ii) for each i, $1 \leqslant i \leqslant n$, either
(a) $F_i \in Ax$,
(b) $\models_{\mathcal{G}} F_i$, or
(c) There exist $j_1, \cdots, j_m$, with $1 \leqslant j_k < i$ for $k = 1, \cdots, m$, such that

$$\frac{F_{j_1}, \cdots, F_{j_m}}{F_i} \in Pr$$

Formal proof with respect to a grammar $\mathcal{G}$ differs from the formal proof defined in Chapter 4 only in that formulae which are valid with respect to $\mathcal{G}$ may be used as if they are axioms. Soundness and completeness clearly take on new meanings under such a system. We still need to know that our system $Ax,Pr,\mathcal{G}$ is sound, that is

$$\vdash_{Ax,Pr,\mathcal{G}} F \supset \models_{\mathcal{G}} F$$

and the completeness of our system:

$$\models_{\mathcal{G}} F \supset \vdash_{Ax,Pr,\mathcal{G}} F$$

is trivial, since F can be used as a premise in any proof in the system $Ax,Pr,\mathcal{G}$. It will not in general be the case, though, that we can construct a proof $\vdash_{Ax,Pr,\mathcal{G}} F$ in which $\models_{\mathcal{U}} F$ is not one of the premises. This is the sort of proof for which we would like to be able to develop a completeness result.

Our completeness proof, Theorem depends on the ability to find the strongest postcondition $sp(K,P)$ for assertion P given code K. We would need to formulate a new kind of postcondition $sp(K,P,\mathcal{G})$ which gives the strongest postcondition in states of $\mathcal{L}_{\mathcal{G}}$ in order to prove the completeness result above. There is no precedent in the literature for such an approach, and we consider the possibility of formalizing a system based on such a postcondition questionable.

Since it is our intent here only to simplify development of proofs, we have no objection to using the system above with the weaker completeness result:

$$\models F \supset \vdash_{Ax,Pr} F$$

Although there may be more compact proofs with respect to $\mathcal{G}$ of F than we can develop without respect to $\mathcal{G}$, this is of no interest to us.

Theorem 5-8: Let Ax and Pr be as in Definition 4-4, $\mathcal{G}$ an h-graph grammar. Then for all $P,Q \in Assn, K \in Code$, if for every procedure call $p(\bar{\alpha})$, $nonprefix(\bar{\alpha})$ holds at the time of call, and for all $\sigma \in \mathcal{L}_{\mathcal{G}}$ and $i, 1 \leqslant i \leqslant |Tcomp(K)(\sigma)|, Tcomp(K)(\sigma)\downarrow i \in \mathcal{L}_{\mathcal{G}}$. then

$$\vdash_{Ax,Pr,\mathcal{G}} \{P\}K\{Q\} \supset \models_{\mathcal{G}} \{P\}K\{Q\}$$

Proof: The validity with respect to $\mathcal{G}$ of the axioms and formulae satisfying

$\models_{\mathcal{G}} F$ is assumed. The soundness of the proof rules is also clear given the conditions above. The result follows by an argument exactly similar to that used in proof of Theorem .

□

Now that we can incorporate valid formulae about program grammars into proofs of correctness formulae we are able to exploit results like the following one. This theorem permits us to remove assertions of the form $eq(s_1,s_2)$ from consideration in proofs of programs where the type structure tells us that s_1 and s_2 cannot possibly select the same types of nodes.

Theorem 5–9:
Given $\mathcal{P}=\langle\sigma,K,\pi,\mathcal{G}\rangle$ a type secure program, and selectors s_1 and s_2. If

$$\theta_{\mathcal{G}}(s_1)\cap\theta_{\mathcal{G}}(s_2)=\varnothing,$$

then

$$\forall i,\ 1\leq i\leq |Tcomp\llbracket K\rrbracket(\sigma)|,\ [\![\neg eq(s_1,s_2)]\!](Tcomp\llbracket K\rrbracket(\sigma)\downarrow i).$$

And given $p\in pnames(\mathcal{P})$, $\pi(p)=\langle\sigma_p,K_p,\bar{\eta}_p,\mathcal{G}_p\rangle$, and selectors s_1 and s_2. If

$$\theta_{\mathcal{G}_p}(s_1)\cap\theta_{\mathcal{G}_p}(s_2)=\varnothing,$$

then

$$\forall i,\ 1\leq i\leq |Tcomp\llbracket K_p\rrbracket(\sigma)|,\ [\![\neg eq(s_1,s_2)]\!](Tcomp\llbracket K_p\rrbracket(\sigma)\downarrow i).$$

Proof: By contradiction.
Let the shorthand σ_i represent $Tcomp\llbracket K\rrbracket(\sigma)\downarrow i$. Suppose $\exists i,\ [\![eq(s_1,s_2)]\!](\sigma_i)$. Then $[\![s_1]\!](\sigma_i)=[\![s_2]\!](\sigma_i)$. Since $\mathcal{P}$ is type secure, we know by Theorem 5–3 that $\tau^*_{\sigma_i}(s_1)\in\theta_{\mathcal{G}}(s_1)$ and $\tau^*_{\sigma_i}(s_2)\in\theta_{\mathcal{G}}(s_2)$. And since $\theta_{\mathcal{G}}(s_1)\cap\theta_{\mathcal{G}}(s_2)=\varnothing$ we deduce that $\tau^*_{\sigma_i}(s_1)\neq\tau^*_{\sigma_i}(s_2)$ But, since $[\![s_1]\!](\sigma_i)=[\![s_2]\!](\sigma_i)$, $\tau^*_{\sigma_i}(s_1)\in\theta_{\mathcal{G}}(s_1)=\tau^*_{\sigma_i}(s_2)\in\theta_{\mathcal{G}}(s_2)$ must hold, a contradiction.

The proof for the case involving procedure calls is analogous.

□

5.3. Introducing Grammars into Hg Programs

In this section we present a syntax for h–graph grammars and show how this is incorporated into Hg programs.

```
h-graph-grammar ::= type-definition
                    ...
                    type-definition

type-definition ::= type-name '::=' rhs

rhs ::= alternative | rhs '|' alternative

alternative ::= bnf-type-name |
                initial-node-arcset
                    node-arcset
                    ...
                    node-arcset

initial-node-arcset ::= node-arcset

node-arcset ::= node-name : [ type-list ]
                    - arc-label -> node-name : [ type-list ]
                    ...
                    - arc-label -> node-name : [ type-list ]

arc-label ::= identifier

type-list ::= type-name | type-list , type-name

type-name ::= identifier

bnf-type-name ::= identifier

h-graph-name ::= identifier

graph-name ::= identifier

node-name ::= identifier
```

Figure 5.2 Syntax for H-graph Grammars.

Figure 5.2 presents a syntax for h-graph grammars, but a few details of that syntax need clarification. The first typename appearing in the grammar is assumed to be the type of the rootgraph. We have omitted the definition of any BNF types in the grammar. The BNF types describe atomic data objects. In a real setting, these primitive types would be language-defined rather than program-defined so a mechanism for their definition within an h-graph grammar is not provided. We also use the convention from Chapter 2 that if a node has a single entering arc and no exiting arcs, then its typelist or value may be

written immediately following the first occurence of its name. A typed h–graph is written in the syntax provided in Figure 5.3.

In writing an Hg program $\langle \sigma, K, \pi, \mathcal{G} \rangle$ we use the following syntax:

```
h–graph ::= h–graph–name : { root–graph
                             graph
                             ...
                             graph }

root–graph ::= graph

graph ::= graph–name : initial–node–arcset
                       node–arcset
                       ...
                       node–arcset

initial–node–arcset ::= node–arcset

node–arcset ::= node–name : [ node–value , type–name ]
                            – arc–label –> node–name : [ node–value , type–name ]
                            ...
                            – arc–label –> node–name : [ node–value , type–name ]

node–value ::= graph–name | atom

arc–label ::= atom

type–name ::= identifier

atom ::= character string

h–graph–name ::= identifier

graph–name ::= identifier

node–name ::= identifier
```

Figure 5.3 Syntax for Typed H–graphs.

program ::= **program**
type
$\mathcal{G}$ <syntax from Figure 5.2 >
var
σ < syntax from Figure 5.3 >
begin
K <syntax from chapter 2 >
end;

and each procedure definition $\langle \sigma_p, K_p, \overline{\eta}_p, \mathcal{G}_p \rangle$ with its name p as given by π, is written:

procedure defn ::= **procedure** p ($\overline{\eta}_p$);
type
$\mathcal{G}_p$ < syntax from Figure 5.2 >
var
σ_p < syntax from Figure 5.3 >
begin
K_p < syntax from chapter 2 >
end;

Now we can finally present a non-trivial program example demonstrating the use of all we have developed so far. We assume the type **integer** is defined to have the usual elements, the type **empty** contains the value #, and type **nil** contains the single value nil.

Example 5-1: (Stack Program)

This is the program of Example 2-2 extended to include type information.

program

type
local-state ::= n_1:[**empty**]
- *value*1 -> n_2:[**integer**]
- *value*2 -> n_3:[**integer**]
- *stack* -> n_4:[stack_type, **nil**]

stack_type ::= n_1:[**empty**]
- *head* -> n_2:[**integer**]
- *tail* -> n_3:[stack_type, **nil**]

var
prog-local-state : { g_1:n_1:[# , **empty**]
- *value*1 -> n_2:[6, **integer**]
- *value*2 -> n_3:[3, **integer**]
- *stack* -> n_4:[nil , **nil**]

}

```
begin
        push( /value1, /stack );
        push( /value2, /stack )
end;

procedure push ( /value, /stk );
type
                state ::= n1:[ empty ]
                                - newelement -> n2:[ stack_type ]
                                - value -> n3:[ integer ]
                                - stk -> n4:[ stack_type, nil ]
var
        push-local-state : {  g1:n1:[ # , empty ]
                                - newelement -> n2:[ g2, stack_type ]
                                - value -> n3:[ 0, integer ]
                                - stk -> n4:[ nil, nil ]

                            g2: n5:[ # empty ]
                                - head -> n6:[ 0, integer ]
                                - tail -> n7:[ nil, nil ]
                    }

begin
        /newelement /head := /value;
        /newelement /tail := /stk;
        /stk := /newelement
end
```

Notice in Example 5–1, that a production for the type stack_type is given only in the main program. This is because the procedure grammar which makes use of the type name stack_type is a co–grammar and uses the same definition, hence that definition need not be repeated.

5.4. Applying Type Information in Program Proofs

We now show how the theoretical results of the preceding sections can be applied to an actual program proof, to help manage the complexity which can arise due to the assignment axiom. The example we use is the stack example 2–2. We extend the example to incorporate initial state grammars as described in the previous section and verify the assertions made use of in Example 4–1.

Suppose we are given the following information about the procedure name *push* in program $\mathcal{P} = \langle \sigma, code, \pi, \mathcal{G} \rangle$:

$$\pi(push) = \langle \sigma_{push}, K_{push}, \overline{\eta}, \mathcal{G} \rangle$$

Suppose we want to prove the following property of the procedure body:

$$\{/value = V \ \wedge \ /stk = S\}$$

$$K_{push}$$

$$\{/stk\,/head = V \ \wedge \ /stk\,/tail = S\}$$

We show how this can be proven in the proof system presented here.

Lemma 5–10: The program of Example 5–1 is statically type secure.

Proof: Notice that the main program will be statically type secure if

$$\theta_{\mathcal{G}}(/value\,1) = \theta_{\mathcal{G}}(/value\,2) = \theta_{\mathcal{G}_{push}}(/value) = \{\textbf{integer}\}$$

and

$$\theta_{\mathcal{G}}(/stack) = \theta_{\mathcal{G}_{push}}(/stk) = \{\text{stack–type},\textbf{nil}\}$$

both of which conditions are clearly satisfied.
The procedure *push* is statically type secure if

$$\{\textbf{integer}\} = \theta_{\mathcal{G}_{push}}(/newelement\,/head) \supseteq \theta_{\mathcal{G}_{push}}(/value) = \{\textbf{integer}\}$$

and

$$\{\text{stack–type},\textbf{nil}\} = \theta_{\mathcal{G}_{push}}(/newelement\,/tail) \supseteq \theta_{\mathcal{G}_{push}}(/stk) = \{\text{stack–type},\textbf{nil}\}$$

and

$$\{\text{stack–type},\textbf{nil}\} = \theta_{\mathcal{G}_{push}}(/stk) \supseteq \theta_{\mathcal{G}_{push}}(/newelement) = \{\text{stack–type}\}$$

all of which are satisfied as well, therefore the program is statically type secure.

□

Lemma 5–11: $\models_{\mathcal{G}} \{/newelement\,/head \neq \perp \ \wedge \ /newelement\,/tail \neq \perp \ \wedge /stk \neq \perp\}$

Proof: Inspection reveals that selectors */newelement /head*, */newelement /tail*, and */stk* are total over $\mathcal{G}$, hence each must select some node in any state in $\mathcal{L}_{\mathcal{G}}$.

□

First we want to show that

$$\vdash_{Ax,Pr,\mathcal{G}} \begin{array}{l} \{/stk = S \wedge /value = V\} \\ /newelement/head := /value; \\ newelement/tail := /stk; \\ /stk := /newelement \\ \{/stk/tail = S \wedge /stk/head = V\} \end{array}$$

To do this, we will use the assignment axiom thrice, making several simplifications (based on grammar $\mathcal{G}$) via the rule of consequence. We derive the proof starting from the postcondition.

First we want to determine the precondition of the statement

$$/stk := /newelement$$

which will result in the truth of

$\{/stk/tail = S \wedge /stk/head = V\}$, that is

$wp(/stk := /newelement, /stk/tail = S \wedge /stk/head = V)$ Applying $\Delta^{/newelement}_{/stk}$, we see that the precondition is

(*)

$$\begin{array}{l} \neg eq(/stk, \perp) \\ \wedge \Big[eq(/stk, /stk) \wedge \\ \quad [eq(/newelement/tail, /stk) \wedge /newelement = S \\ \quad \vee \neg eq(/newelement/tail, /stk) \wedge /newelement/tail = S] \\ \quad \vee \neg eq(/stk, /stk) \wedge eq(/stk/tail, /stk) \wedge /newelement = S \\ \quad \vee \neg eq(/stk, /stk) \wedge \neg eq(/stk/tail, /stk) \wedge /stk/tail = S \Big] \\ \wedge \Big[eq(/stk, /stk) \wedge \\ \quad [eq(/newelement/head, /stk) \wedge /newelement = V \\ \quad \vee \neg eq(/newelement/head, /stk) \wedge /newelement/head = V] \\ \quad \vee \neg eq(/stk, /stk) \wedge eq(/stk/head, /stk) \wedge /newelement = V \\ \quad \vee \neg eq(/stk, /stk) \wedge \neg eq(/stk/head, /stk) \wedge /stk/head = V \Big] \end{array}$$

While the above tells us what precondition is true given any state, we can make certain simplifications in the event that we know that the state is in $\mathcal{L}_{\mathcal{G}}$. And since we know that any state σ reachable in program execution of procedure *push* is in $\mathcal{L}_{\mathcal{G}}$, it is reasonable to use this information to our advantage. In particular, Theorem 5–4 tells us that the following three assertions are true in any state the procedure above can reach:

$$eq(/stk, /stk)$$

$$\neg eq(/newelement/tail, /stk)$$

$$\neg eq(/newelement, /newelement/tail)$$

and Lemma 5–11 insures that

$$\neg eq(/stk, \perp)$$

And we can simplify this precondition (*) via the rule of consequence with the following fact:

$$\models_{\mathcal{G}} \quad /newelement/tail = S \wedge /newelement/head = V \supset (*)$$

The precondition for the second assignment:

$$wp(/newelement/tail := /stk, /newelement/tail = S \wedge /newelement/head = V)$$

is derived by applying the assignment axiom again. This gives us the following precondition:

$$\neg eq(/newelement\,/tail, \perp) \qquad (**)$$

$$\wedge \Big[eq(/newelement, /newelement\,/tail) \wedge$$

$$[eq(/stk\,/tail, /newelement\,/tail) \wedge /stk = S$$

$$\vee \neg eq(/stk\,/tail, /newelement\,/tail) \wedge /stk\,/tail = S]$$

$$\vee \neg eq(/newelement, /newelement\,/tail) \wedge eq(/newelement\,/tail, /newelement\,/tail) \wedge /stk = S$$

$$\vee \neg eq(/newelement, /newelement\,/tail) \wedge \neg eq(/newelement\,/tail, /newelement\,/tail)$$

$$\wedge /newelement\,/tail = S \Big]$$

$$\wedge \Big[eq(/newelement, /newelement\,/tail) \wedge$$

$$[eq(/stk\,/head, /newelement\,/tail) \wedge /stk = V$$

$$\vee \neg eq(/stk\,/head, /newelement\,/tail) \wedge /stk\,/head = V]$$

$$\vee \neg eq(/newelement, /newelement\,/tail) \wedge eq(/newelement\,/head, /newelement\,/tail) \wedge /stk = V$$

$$\vee \neg eq(/newelement, /newelement\,/tail) \wedge \neg eq(/newelement\,/head, /newelement\,/tail) \wedge$$

$$/newelement\,/head = V \Big]$$

By Theorem 5–4, we know that

$$\neg eq(/newelement, /newelement\,/tail)$$

and clearly

$$eq(/newelement\,/tail, /newelement\,/tail)$$

and since $\theta_{\mathcal{G}_{push}}(/newelement\,/head) \cap \theta_{\mathcal{G}_{push}}(/newelement\,/tail) = \varnothing$, Theorem 5–9 tells us that

$$\neg eq(/newelement\,/head, /newelement\,/tail)$$

and Lemma 5–11 guarantees

$$\neg eq(/newelement\,/tail, \perp)$$

so we can employ the following formula in this instance:

$$\models_g \{/stk = S \wedge /newelement/head = V \supset (**)\}$$

Note that we could not have derived this using Theorem 5–4 alone, since that can only guarantee that selectors of nodes in the root graph cannot be aliased. Since selectors selecting other nodes through the root graph are potentially aliased, we must resort to some other means of determining whether it is the case that they cannot be aliased with other nodes. In this case the selectors */newelement/head* and */newelement/tail* might be aliased with each other, but because we know they appear in a statically type secure program and their typesets are disjoint, they cannot be aliased. Were the procedure *push* not statically type secure, we might not be able to make this argument. To do so, we would have to formally prove that no execution sequence which could lead to a state which was not type correct. Since generation of such proofs is an undecidable problem, it is probably not wise to rely on this as a method of simplifying program proofs.

The precondition

$$wp(/newelement/head := /value, /stk = S \wedge /newelement/head = V)$$

is gotten once again by applying the assignment axiom. This yields the following assertion:

$$\begin{aligned}
&\neg eq(/newelement/head, \perp) \qquad (***)\\
&\wedge \Big[eq(/stk, /newelement/head) \wedge /value = S\\
&\quad \vee \neg eq(/stk, /newelement/head) \wedge /stk = S\Big]\\
&\wedge \Big[eq(/newelement, /newelement/head) \wedge\\
&\quad [eq(/value/head, /newelement/head) \wedge /value = V\\
&\qquad \vee \neg eq(/value/head, /newelement/head) \wedge /value/head = V]\\
&\quad \vee \neg eq(/newelement, /newelement/head) \wedge eq(/newelement/head, /newelement/head) \wedge /value = V\\
&\quad \vee \neg eq(/newelement, /newelement/head) \wedge \neg eq(/newelement/head, /newelement/head) \wedge
\end{aligned}$$

$/newelement\,/head = V$ ⌋

Once again, by Theorem 5–4

$$\neg eq(/newelement\,,/newelement\,/head\,)$$

$$\neg eq(/stk\,,/newelement\,/head\,)$$

and clearly

$$eq(/newelement\,/head\,,/newelement\,/head\,)$$

and by Lemma 5–11 we know that

$$\neg eq(/newelement\,/head\,,\perp)$$

so we know

$$\models_{\mathcal{G}} \{/stk = S \wedge /value = V \supset (***)\}$$

Restating the information above as a theorem, we have the following:

Theorem 5–12:

$$\vdash_{Ax,Pr,\mathcal{G}} \begin{array}{l} \{/stk = S \wedge /value = V\} \\ /newelement\,/head := /value; \\ /newelement\,/tail := /stk; \\ /stk := /newelement \\ \{/stk\,/tail = S \wedge /stk\,/head = V\} \end{array}$$

Proof:

$$\{\neg eq(/newelement\,/head\,,\perp) \wedge \neg eq(/newelement\,/tail\,,\perp) \wedge \neg eq(/stk\,,\perp) \wedge /stk = S \wedge /value = V \supset (***)\}$$

By Lemma 5–11.

$$\{(***)\}$$
$$/newelement\,/head := /value$$
$$\{/stk = S \wedge /newelement\,/head = V\}$$

Assignment axiom and rule of consequence.

$$\{/stk = S \wedge /newelement\,/head = V \supset (**)\}$$

Valid with respect to $\mathcal{G}$.

$$\{(**)\}$$
$$/newelement\,/tail := /stk$$
$$\{/newelement\,/tail = S \wedge /newelement\,/head) = V\}$$

Assignment axiom and rule of consequence.

$$\{/newelement/tail = S \wedge /newelement/head) = V \supset (*)$$

Valid with respect to $\mathcal{G}$.

$$\{(*)\}/stk := /newelement\{/stk/tail = S \wedge /stk/head = V\}$$

Assignment axiom and rule of consequence.

Our result follows immediately from several applications of the rule of consequence and the rule of sequential composition.

□

5.5. Chapter Summary

In Section 1 we presented the definitions of an h–graph grammar and typed h–graph. We introduced the selector typeset function θ, and gave an example of its computation. We extended the definition of the language Hg to incorporate the notion of data type as reflected in a typed h–graph.

We defined the properties of type security and static type security in Section 2. We showed that in every case a statically type secure program is type secure and that for at least one class of programs static type security and type security are equivalent. We also demonstrated that selectors with disjoint typesets cannot be aliased in a statically type secure program. An example of the simplification of a proof of a stack manipulation program is given in Sections 3 and 4 to illustrate the application of these ideas.

Chapter 6

Conclusion

We have presented a syntactic and semantic definition of the programming language Hg along with a formal system of verification for Hg programs. In this chapter we summarize the major results of this work. We then discuss possible extensions to the work and look at some other issues which arise concerning the language Hg and its proof system.

6.1. Summary of Major Results

The basic definitions of an h–graph and h–graph selector were given by Pratt [6], who outlined the first version of Hg, as well. In Chapters 2 and 5, we rigorously formalize these basic concepts to provide a framework for the formal definition of Hg. This language has much of the power which might be found in a language like Pascal, but with a very clean mathematical structure. The language supports, for example, complex data objects which can model arrays, records, and pointers; typical control structures like conditionals and while loops; procedures with parameters; uniform assignment of all data objects, permitting asssignment of pointer objects; dynamic allocation of storage; and type definitions with more generality than those found in Pascal, for instance.

We provide in Chapters 2 and 5 a complete, clear, and concise definition of Hg, which constructively defines the state sequence which results from the execution of any Hg program. This definition reflects in a straightforward manner the change of referencing environment during procedure calls, and gives the semantics of data structures and references, dynamic storage allocation, assignment, and all of the control structures of the language Hg.

In Chapter 3 we present an assertion language for Hg and define the meanings of these assertions in terms of our language definition and an underlying logic. This step, which is often taken for granted, is crucial to developing a correct verification system for a programming language. We also develop an assignment axiom scheme for Hg assignments which correctly provides the weakest precondition for any assignment and postcondition. This assignment axiom works for assignments involving structure components and aliased selectors in arbitrarily complex data structures, even those involving aliased circular selector references. This axiom is more general than that of any previous work and represents an important advance in understanding the behavior of programs which involve pointer chains.

In Chapter 4 we develop a sound and complete verification system for Hg based on the assignment axiom, with proof rules for all the control structures of Hg and for non-recursive procedure calls with some restrictions on the arguments. The system is complete for the Hg programs considered, meaning that as much as can be proven under any system is proven under ours. Although our procedure call rule makes some restrictions on the parameters in the call, these restrictions are reasonable ones, and the proven completeness of the rule makes it more desirable than one which, though more powerful, may be incorrect. The extension of the procedure call rule to one which can be applied to recursive procedures is relatively straightforward.

We also develop a new theory of data types and type checking based on Pratt's notion of using h-graph grammars to define the data types of a program [4]. In Chapter 5 we formalize h-graph grammars for use in type definitions and develop a way of associating this grammar based type information with nodes in h-graphs and selectors in a program. This model of data types lets us give a formal meaning to the concept of type correctness of a program state and provides us with a means of checking type correctness of programs, both dynamically and statically. We present a method of using program

type information to simplify formal proofs of correctness of Hg programs by eliminating those parts of assertions which account for aliases which can never exist. We prove that this method of simplifying program proofs is correct. No other work known to the author has presented a definition of type correctness of programs based on a formal model of data types, and we know of no work which incorporates information derived from the data type structure of a program into program proofs.

In short, the work presented here represents a major advance of the study of formal semantics and program verification into the realm of languages which support complex data objects, programmer defined data types, and assignments and procedures which manipulate these complex objects.

6.2. Possible Extensions to this Work

Although the language Hg is powerful, we have omitted a number of constructs which are common to many languages. We summarize here what some of those constructs might be.

Although the mechanism for providing arrays exists in the data structures of Hg through h–graphs, we present no method of accessing arrays in expressions by using *computed subscripts*. This is not for want of a reasonable method of defining the semantics of computed subscripts or because deriving proof rules in such an environment is impossibile; but the extra level of complexity involved was felt to be prohibitive for the current work.

No proof technique is provided for procedure calls in which there are aliases in the actual parameters. The abstract complexity of the procedure call rule provided here is undeniable. The author has a scheme for developing a rule for procedure calls with arbitrarily aliased formal and actual parameters which he feels to be correct, but he has yet to be able to demonstrate the correctness of such a rule. This is in part due to the complexity of the assignment axiom upon which the proposed rule is based. It is not clear

whether providing such a procedure call rule is worth the effort which its development would require. Extension of the procedure call rule to permit recursive procedures would be relatively straightforward.

The language Hg does not incorporate a mechanism for passing procedures as parameters to other procedures. Such an extension to Hg is possible, since our semantics provides a domain, *Proc*, which contains objects encoding the meanings of procedures. The complexity of program proof in such a system, on the other hand, is questionable.

There are no programmer definable functions in Hg. In addition, Hg programmers cannot define new boolean operations. These facilities were consciously omitted from the language. The semantic definition of Hg is quite straightforward as it stands. The procedure call introduces more complexity than any other statement due to its introduction of new nodes into the state. To burden assignments and condition evaluation with introducing nodes into the state graph was felt to be too troublesome.

Hg is a scopeless language. We felt the addition of the typical Pascal style scope rules to the language would be a mistake. Our definition of procedure calls requires that selectors in a procedure select nodes which are accessible from the root graph of the procedure calling state. Pascal scoping would require us to be able to select nodes starting from graphs in the state which are inaccessible from the root graph. This in itself is not difficult, but it would add one more level of complexity to the language definition. We feel that the advantages and disadvantages should be weighed carefully before introducing scopes into Hg.

We have not considered programs in which execution can take place on parallel paths. This would require a major rethinking of the semantics of the language, since the function *Comp*, the semantics defining function for Hg, requires that a program produce an ordered sequence of states. Stotts presents a Petri–Net based method for analyzing h–graph computations in a concurrent processing environment [38].

6.3. Other Questions of Interest

Proofs of programs involving aliases can become quite complex. The example of Chapter 3 shows that even a single assignment involving two selectors can generate a precondition with numerous terms. Two approaches to verification of larger Hg programs immediately come to mind:

(i) restrict the language to make verification easier, or

(ii) provide automatic verification tools for Hg.

It is clear that further restrictions on Hg could simplify the task of program verification. Elimination of some kinds of aliases could permit our assignment axiom to make use of one of the simpler substitution rules presented in Chapter 3. It is not clear that such simplifications are desirable however.

A verification condition generator (which could provide preconditions for statements mechanically) would be quite useful. The author could have used such a tool several times during the preparation of this work. Such a program should not be difficult to produce. The problem of providing a general theorem prover for the assertion language we have presented is a more difficult question. Additional thought would need to be given to the relation of the assertion language to the underlying logic before such a task could be undertaken.

Hg is a powerful yet simple language, but one may wonder if it could be implemented and used for writing real programs. Our immediate answer is that yes, Hg could be implemented but one must be careful in doing so. Recall from Chapter 2, for example, that each procedure call uses fresh nodes from the universe of nodes and leaves some garbage in the calling state. We have not provided any formal basis for collection of this garbage, though it is clear that such a basis could be developed with the tools we have provided.

We feel that Hg serves as a prototype for development of a new class of programming languages having simple, regular syntactic and semantic structure, but the power of languages of the Pascal class. A prototype implementation of Hg in Lisp has demonstrated that small Hg programs can be developed even when a tortuous programmer interface is provided. The data structuring power provided by h-graphs suggests development of a Lisp-like language based around directed graphs rather than lists.

The user of Hg, or any other programming language for that matter, can benefit not only from clean semantic definitions like that provided here, but also from programming tools such as type checkers, verification condition generators, and program transformers based on such formal models.

References

1. R. Cartwright and D. Oppen, "The Logic of Aliasing," *Acta Informatica*, **15** (1981), 365–384.

2. E.–R. Olderog, "Sound and Complete Hoare–like Calculi Based on Copy Rules," *Acta Informatica*, **16** (1981), 161–197.

3. T. W. Pratt, "A Hierarchical Graph Model of the Semantics of Computer Programs," *Proceedings of the Spring Joint Computer Conference* (1969), 813–825.

4. T. W. Pratt, "A Theory of Programming Languages, Part I," Report CCSN–41, University of Texas Computation Center, Austin (1975).

5. T. W. Pratt, "Application of Formal Grammars and Automata to Programming Language Definition," in R. T. Yeh(ed.), *Applied Computation Theory*, Prentice–Hall (1976).

6. T. W. Pratt, "Formal Specification of Software Using H–Graph Semantics," in *Lecture Notes in Computer Science #153: Graph Grammars and Their Application to Computer Science*, Springer–Verlag (1983), 314–332.

7. D. Gries, *The Science of Programming*, Springer–Verlag, New York (1981).

8. R. L. London, J. V. Guttag, H. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek, "Proof Rules for the Programming Language Euclid," *Acta Informatica*, **10**(1) (1978), 1–26.

9. C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal," *Acta Informatica*, **2** (1973), 335–355.

10. S. Hantler and J. King, "An Introduction to Proving the Correctness of Programs," *ACM Computing Surveys*, **8**(3) (September 1976), 331–353.

11. H. K. Berg, W. E. Boebert, W. R. Franta, and T. G. Moher, *Formal Methods of Program Verification and Specification*, Prentice–Hall, Englewood Cliffs, New Jersey (1982).

12. Z. Manna, *Mathematical Theory of Computation*, McGraw–Hill, New York (1974).

13. C. A. R. Hoare and P. E. Lauer, "Consistent and Complementary Formal Theories of the Semantics of Programming Languages," *Acta Informatica*, **3** (1974), 135–153.

14. J. E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Massachusetts (1977).

15. S. A. Cook, "Soundness and Completeness of an Axiom System for Program Verification," *Siam Journal of Computing*, **7**(1) (February 1978), 70–90.

16. P. Wegner, "The Vienna Definition Language," *ACM Computing Surveys*, **4**(1) (March 1972), 5–63.

17. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *LISP 1.5 Programmer's Manual*, M.I.T. Press, Cambridge, Massachusetts (1965).

18. J. McCarthy, "A Basis for a Mathematical Theory of Computation," in C. M. Popplewell(ed.), *Information Processing 1962 (Proceedings of the IFIP Congress, 1962)*, North–Holland (1963).

19. T. W. Pratt, "H–Graph Semantics," DAMACS Technical Reports #81–15, #81–16, University of Virginia, Charlottesville, Virginia (1981).

20. T. W. Pratt, "Pair Grammars, Graph Languages, and String–to–Graph Translations," *Journal of Computer and System Sciences* (December 1971), 560–595.

21. C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, **12**(10) (October 1969), 322–329.

22. A. R. Meyer and J. Y. Halpern, "Axiomatic Definitions of Programming Languages: A Theoretical Assessment," *Journal of the ACM*, **29**(2) (April 1982), 555–576.

23. K. R. Apt, "Ten Years of Hoare's Logic: A Survey––Part 1," *ACM Transactions on Programming Languages and Systems*, **3**(4) (October 1981), 431–483.

24. M. J. O'Donnell, "A Critique of the Foundations of Hoare Style Programming Logics," *Communications of the ACM*, **25**(12) (December 1982), 927–935.

25. E. W. Dijkstra, *A Discipline of Programming*, Prentice–Hall, Englewood Cliffs, New Jersey (1976).

26. I. Greif and A. R. Meyer, "Specifying the Semantics of While Programs: A Tutorial and Critique of a Paper by Hoare and Lauer," *ACM Transactions on Programming Languages and Systems*, **3**(4) (October 1981), 484–507.

27. J. W. de Bakker, *Mathematical Theory of Program Correctness*, Prentice–Hall International, Englewood Cliffs, New Jersey (1980).

28. Herbert B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, New York (1972).

29. R. M. Burstall, "Some Techniques for Proving Correctness of Programs Which Alter Data Structures," in *Machine Intelligence 7*, John Wiley and Sons, Toronto (1972).

30. S. Owicki and D. Gries, "Axiomatic Proof Techniques for Parallel Programs," *Acta Informatica*, **6** (June 1976), 319–340.

31. S. C. Kleene, *Introduction to Metamathematics*, D. Van Nostrand Company, Inc., Princeton, New Jersey (1952).

32. Nicholas Rescher, *Many-Valued Logic*, McGraw-Hill, New York (1969).

33. D. C. Oppen and S. A. Cook, "Proving Assertions About Programs That Manipulate Data Structures," *Proceedings of Seventh Annual Symposium on Theory of Computing* (May 1975), 107-116.

34. J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in R. T. Yeh(ed.), *Current Trends in Programming Methodology, Vol. IV: Data Structuring*, Prentice-Hall, Englewood Cliffs, N.J. (1978).

35. J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, **10** (1978), 25-52.

36. T. W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, New Jersey (1984).

37. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Massachusetts (1979).

38. P. D. Stotts, Jr., "A Hierarchical Graph Model of Concurrent Software Systems," Ph. D. Dissertation, Department of Computer Science, University of Virginia, Charlottesville, Virginia (May 1985).