A Logic for the Russell Programming Language

Hans-Juergen Boehm

Ph.D Thesis

TR 84-593
February 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

now at:
Department of Computer Science
FR-35
University of Washington
Seattle, Washington 98195

A LOGIC FOR THE RUSSELL PROGRAMMING LANGUAGE

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Hans-Juergen Karl Hermann Boehm

January 1984

A LOGIC FOR THE RUSSELL PROGRAMMING LANGUAGE

Hans-Juergen Karl Hermann Boehm, Ph.D.

Cornell University 1984

We consider a programming language with a number of characteristics detrimental to conventional axiomatic descriptions. These include arbitrary side effects in expressions, aliasing among variables, very general recursive function declarations, and the ability to pass functions as parameters and return them as results. We give an axiomatic definition of this language based on a novel formalism. We prove the axiomatization sound and relatively complete with respect to a (somewhat nonstandard) denotational semantics. In spite of the nonstandard formalism, most conventional techniques for developing and reasoning about programs can be carried over.

This approach leads to different perspectives on several issues. Neither the total nor partial correctness treatments of nonterminating programs are acceptable in this context. Arrays can be treated directly as sequences of variables. The fact that arbitrary "statements" can be embedded in expressions makes the whole development simpler, rather than more difficult. Assignment can be described very easily, in spite of the presence of aliasing and arbitrary expressions on the left side.

On the negative side, nondeterminism becomes a much more difficult issue. The logic seems ill-suited to reasoning about concurrency.

It has frequently been argued that programming languages should be designed so that they can easily be defined axiomatically. These results point out that this criterion is highly dependent on the particular formal system chosen.

BIOGRAPHICAL SKETCH

Hans-Jürgen Karl Hermann Böhm was born in Kiel (Federal Republic of Germany) on August 28, 1956. He is the son of Karl-Heinz Böhm and Erika Böhm-Vitense. After a move to Heidelberg (W. Germany) and several extended visits to Berkeley (California), his family moved to Seattle in 1968.

Mr. Böhm attended junior high school and high school in Seattle. He received B.S. degrees in Mathematics and Computer Science from the University of Washington in 1978. In the course of his study he spent one semester at the University of Göttingen (W. Germany).

Mr. Böhm received an M.S. Degree (Computer Science) from Cornell University in 1980, and is currently on the faculty at the University of Washington.

ACKNOWLEDGEMENTS

I would like to thank Bob Constable and especially Alan Demers for the many contribution they made to this thesis, from providing me with much of the background information, through their comments on the many incomplete drafts of this thesis.

This thesis benefited greatly from comments on the preliminary versions of this material presented at POPL 82 and at a number of other talks. Mike O'Donnell first made me aware of the fact that the value of a variable at a certain point in a computation is also expressible in Dynamic Logic. Ravi Sethi pointed out the similarities between the system presented here and standard denotational semantics. Leslie Lamport suggested some of the difficulties with nondeterministic constructs. He and David Harel referred me to [Mir 71].

Chris Buckley made it possible for me to submit this thesis while I was almost 3000 miles away from Cornell. He deserves thanks for his comments on the final draft as well.

Finally, I would like to thank everyone for their patience as other projects were delayed in the process of completing this thesis. Particular thanks go to the other faculty members at the University of Washington, and to Soroor Ebnesajjad, my fiancée.

TABLE OF CONTENTS

THEOREMS

DEFINITIONS

AXIOMS AND INFERENCE RULES

CHAPTER 1

INTRODUCTION

Since McCarthy (e.g. [McC 63]), Hoare (see [Hoa 69]), and others, did the original work on formal verification of programs, there has been great interest in the definition of programming language constructs through formal inference rules.

Such interest has considerable justification. Formal proof systems provide a precise way of specifying the semantics of such constructs. Such an axiomatic semantic specification of a programming language is directly usable for formal reasoning about programs written in the language. Thus it can serve as a basis for a machine assisted verification system for the language (cf. [Con 78] and [Boy 79]). It may also be used as a foundation for less formal programming methodologies, such as the one presented in [Dij 76] and [Gri 81].

Unfortunately it has been difficult to give axiomatic semantics for most nontrivial programming languages. Few "real" programming languages have been so specified. Even these specifications are sufficiently complex or ill-understood to cast doubt on their correctness. (See for example Gries' discussion of the Euclid procedure call rule in [Gri 80] and O'Donnell's comments on defined functions and "goto" statements [O'Do 82].)

It has been suggested that the difficulty in obtaining axiomatic semantics for conventional languages is an indication of the deficiencies of these languages. A "good" language design should make it easy both to describe the semantics of the language to the programmer and for the programmer to reason about programs writ-

ten in that language. This simplicity should be reflected in a simple formal definition of the language.

The axiomatizability of a programming language and of programming language features has thus been proposed as a criterion for the language designer. It has been used in support of Dijkstra's language [Dij 76], Euclid [Pop 77], and sometimes Pascal [Wir 74]. Pascal and Euclid have usually been characterized with a style of axiomatic system directly derived from Hoare's original work. Dijkstra uses a different notation, but his axioms are still very similar to Hoare's.

We will argue that such an evaluation of programming language features based on their axiomatizability needs to be viewed with caution.

There are few formal results on this subject. It is fairly easy to convince oneself that any programming language could be formally axiomatized if one did not care about the aesthetics of the axiomatization. It would be possible to just model an interpreter for the language. Thus the problem is interesting only because we want an axiomatization which is compact and easy to understand and work with. Needless to say, these criteria are hard to formalize.

The only major negative result dealing with the axiomatizability of programming languages is given in [Cla 76]. Even it is of real interest only when we are dealing with formal systems very similar to Hoare's. Conclusions about "easy axiomatizability" of certain constructs are thus almost exclusively empirical results based on actual attempts to do so. These attempts have almost always been based on Hoare's work.

What follows is a different development of a programming logic. It is used to give an axiomatic semantics of much of Russell [Boe 80], a language that differs substantially in character from languages like Pascal. It has a number of characteristics that have been believed in the past to make a language difficult to axiomatize. Nonetheless in this development they appear to be very natural. Besides giving a development which is of interest in its own right, we thus give some evidence that previous informal conclusions on "easy axiomatizability" are based largely on the particular logic that was used.

We will be dealing primarily with the following three characteristics of Russell:

1. Russell is an expression language. A language such as Pascal distinguishes between statements and expressions. Statements are executed purely for their effect on the state of the machine, whereas expressions generally do not alter the state, but just produce a value. Expression languages such as Algol 68 and Russell do not make this distinction. The only unit in the language is an expression which both yields a value and possibly affects the state.

   There has been some previous work on axiomatizing such languages. (See [Cun 76], [Kow 77], [Pri 77], [Schw 78].) All of this work has been in the form of extensions to Hoare logic which explicitly allow one to talk about the value produced by an expression as part of the state of the computation. Thus it leads to logics which are necessarily more complex than the original Hoare logic. We instead present a development which is only made possible by the expression language.[1]

2. Russell distinguishes explicitly between variables and values. Variables (or locations if you prefer) are objects which may be passed as parameters and returned as results of functions just as values (and functions) may be. Identifiers may be

[1] An earlier and somewhat less formal version of this aspect of the logic is given in [Boe 82].

bound to either variables or values. The binding of an identifier remains fixed inside its scope. An assignment changes the value asssociated with a given variable, but not the variable object denoted by some identifier.

   This distinction is preserved in the logic. Thus evaluation of an expression does not change the object to which an identifier refers. State information is obtained using the ValueOf function, which maps variables to their values. Explicitly dealing with variables as objects in the logic facilitates both the axiomatization of function calls and the treatment of structured data objects. We do not need to introduce additional logical primitives to talk about programming language constructs such as arrays. Within the framework for an expression language logic this also makes it possible to reason about aliased variables with no additional complexity in the logic.

3. Functions (and types) are objects just as values and variables are. Thus the axiomatization of functions and function calls is sufficiently general to talk about higher order functions. This is accomplished within a fairly simple logic by taking a different approach to proof rules for declarations and function calls. Our proof rules explicitly translate argument parameter binding and declaration binding to analogous variable bindings in the predicate calculus. Recursive declarations complicate this somewhat, but we do include rules to reason about them.

Chapter 2 gives an overview of the subset of the Russell language we will be discussing initially. We will also give formal denotational semantics for this language. These semantics are used in the soundness proofs of chapter 4 and occasionally in chapter 5. All of these can be skipped as a unit without loss of continuity.

An unusual style of denotational semantics is used in order to make the treatment consistent with the logic. We obtain fairly elegant definitions of most language constructs in this way. Only the definition of recursive declarations stretches the approach close to its breaking point.

Chapter 3 presents the basic framework of the logic.

Chapter 4 presents proof rules for the language and proofs of their soundness.

Chapter 5 gives a proof that the rules presented are in some sense sufficient for reasoning about the given programming language. This is formally phrased as a relative completeness theorem similar to that of [Coo 76]. The presence of this chapter complicates some of the development. There are cases (related to the treatment of variables in particular) where this kind of result imposes requirements on the logic which do not appear to have much practical motivation. Unfortunately it appears easier to modify the logic to satisfy a theorem in the style of [Coo 76] than to find a restatement which does not impose these restrictions. We will point out those parts of the development inspired by such considerations.

Chapter 6 discusses the applicability of the logic both to the full Russell language and to other programming languages, as well as drawing some general conclusions. A discussion of our rather unorthodox treatment of nontermination is deferred to this chapter.

Note that the main characteristic of Russell, namely its type structure, is almost irrelevant to this discussion. It will be discussed as little as possible until chapter 6. We are really interested in any language which exhibits some of the above characteristics. Our Russell subset serves as a nice illustration.

CHAPTER 2

THE RUSSELL SUBSET

Every syntactic construct in the language which we will describe is an expression. Thus we will describe the language by giving a collection of possible forms which an expression may take.

In the Russell language, as defined in [Boe 80], each expression has a syntactic type, or *signature*, associated with it. We begin with a very brief outline of a gross simplification of this type structure. Since we are interested in semantics and not type structure we will ignore this aspect of the language to the greatest possible extent; however the reader should keep in mind that identifiers in particular are intended to be typed.

2.1. Type Structure

In this simplified form of the language there are no type valued expressions. For the present, we deal with only a finite given collection of types. Any preconceived notion the reader may have as to the precise meaning of the word "type" will do. We will assume only that there is some, not necessarily distinct, collection of values R_T associated with each type T. If an expression yields an r-value (i.e. a value and not a variable) of type T we say it has signature

val T

Such an expression must return an element of R_T. But since the sets R_T are not assumed disjoint, a signature cannot be a property of a value. Thus the signature is a property of the expression itself, and not of the value produced by it; the value may be in several of the R_T. For example, we may take

$$R_{Integer} = \{ ..., -1, 0, 1, 2, ... \}$$

$$R_B = \{ 0, 1 \}$$

B here stands for "Boolean". The expression $a \vee b$ would have signature val B, in spite of the fact that the result is always in $R_{Integer}$ as well. The signature tells us that it does not make sense to use the expression in a context where an integer is required, even though the result could be (mis)interpreted as such.

Our language also includes function and variable valued expressions.

A variable (l-value) can be thought of as simply a location in which a value (r-value) can be stored. If an expression yields an l-value which can contain a value of type T we say it has signature

var T

A simple expression with **var** T signature is

NewT[]

Thus 'NewInteger' is a primitive function which takes no arguments and returns a new, that is previously inaccessible, location to be used for storing integers. A new variable identifier is typically declared by binding the identifier to the result produced by such an expression.

We obtain the value stored at a particular location through the builtin functions TV. For example if x has signature **var** Integer, the expression IntegerV[x] will produce the value stored at location x.

If an expression yields a function which maps the results of expressions with signature T to values which should be interpreted as described by signature S, we say

the expression itself has signature

func[T] S

If the function produced expects n arguments with signatures $T_1, ..., T_n$ it itself has signature

func[$T_1, ..., T_n$] S

Again a function is declared by binding an identifier to a function valued expression. Thus we can declare an Integer variable x, and define a function f to be $g \circ h$ with the two declarations

```
x == NewInteger[];
f == g o h
```

Here we have assumed $\circ$ to be a previously defined operator.

Neither functions nor other l-values can be assigned to variables[1].

We assume that the Russell import rule is obeyed. It requires that no function valued expression may mention a non-local identifier with **var** signature. Thus such an expression may neither depend on, nor change the values of variables known in the scope in which it appears. This is a syntactic property which is part of the definition of type-correctness. It insures that meaning of function valued expressions is completely independent of the state of the computation. (For more details see [Boe 80].) It follows that function producing functions may not have side effects[2].

---

[1] This is true in [Boe 80] to enable the development of the signature calculus used there. It appears plausible that the treatment here could be generalized to allow at least function assignment. That would however require much more complicated domain constructions later in this chapter. Further, some of the techniques used in chapters 4 and 5 presently require that there are only finitely many distinct **var** signatures. This would become false if we allowed either function or l-value valued variables.

[2] Technically, we can write function producing functions with **var** parameters. However there is no legal way to apply these functions. We henceforth ignore them. Note also that functions produced by an application of another function may have side effects.

Since function bodies may not mention global variables, function applications are also side effect free, that is they do not effect the value of the state, unless they have at least one parameter with **var** signature. We say that a function signature, or a function with such a signature, is state dependent if it has **var** parameters.

2.2. Semantic Odds 'n Ends

To formally describe the meaning of expressions we need a few preliminaries. We assume that for each type T we are given a corresponding set R_T, which represents the "possible" values of type T. In particular we have a set R_B consisting of the truth values **true** and **false**. Unlike conventional approaches to denotational semantics (see e.g. [Sto 77] or [deB 80]) we do not assume that R_T has any structure. We treat it simply as a set. Let R be the union of the R_T. Thus all r-values are in R. We similarly define L_T to be the set of all l-values of type T. We assume that the L_T are countably infinite. This will guarantee that we never run out of new l-values. N, the set of non-negative integers will do fine for each of these.

For each L_T we define an associated storage allocation function A_T which for each integer i gives us the i^{th} element of L_T to be allocated[3]. We insist that these functions be 1-to-1 and onto. (Since L_T is countably infinite such functions will always exist.) They will be used to define the semantics of the functions NewT.

We define L to be the disjoint union of the L_T.[4]

---

[3]Our formal model assumes that inaccessible storage is not reclaimed. The fact that a real implementation would do so should disturb the reader only mildly, since such a scheme should not affect the visible behavior of a program.

[4]The fact that the L_T are treated as disjoint is of technical use. In particular it facilitates our simple-minded approach to storage allocation. It has little practical consequence since it doesn't affect the visible behavior of a program.

We postpone discussion of structured data objects, and structured l-values in particular until chapter 6. Thus two l-values are disjoint or identical. Assigning to one either does not affect the other, or is equivalent to assigning to it.

We denote by D_{sig} the domain of "possible" values produced by an expression with signature sig.[5]

Thus we inductively define:

$$D_{\mathbf{val}\,T} = R_T$$

$$D_{\mathbf{var}\,T} = L_T$$

$$D_{\mathbf{func}[s_1, \ldots, s_n]s_0} = (S \times D_{s_1} \times \ldots \times D_{s_n}) \to (D_{s_0} \times S)$$

Here S denotes the set of possible states. It is defined in detail below. As usual $S_1 \to S_2$ denotes the set of functions from S_1 to S_2, and $S_1 \times S_2$ denotes their Cartesian product. Let D, the domain of all programming language objects, be $\cup_{sig} D_{sig}$, that is the union of domains associated with all signatures.

We formally define the meaning of a language construct in the context of an environment $e \in E$. e specifies current bindings for all identifiers in the program (but not values stored in variables.) Thus if I is the set of possible identifiers,

$$E = I \to D$$

We use the notation $I \to D$ to denote the set of all functions mapping I to D.

Each expression can then be viewed as mapping an initial state to a final state and a value. This state now consists of the mapping between variables and values.

---

[5]L_T, R_T, and D_{sig} will usually be strict supersets of the values which can be produced by programming language expressions. They are collections of values which are in some sense plausible. E.g. for the domain of functions produced by expressions with signature func[val T]val T we will take the set of all such functions, rather than just the computable ones.

Thus the domain of possible states S is given by

$$S = (L \to R) \times (T \to N)$$

The $L \to R$ component of the Cartesian product defines a mapping from any location to a value. The second component specifies how many elements of each set L_T have already been allocated, that is have already been returned by some call to NewT[]. More precisely, T is the set of type names; the second component of a state maps the type name T to the non-negative integer i if i elements of L_T have already been allocated.

Given a state s, we refer to its first component as s_m and its second component as s_a.

We can then formally define the meaning of an expression a by associating a function

$$M[\![a]\!] : E \to (S \to S \times D)$$

with it.

Here and in the remainder of this thesis the letters a,b,c,d (and occasionally the greek letters α, β, ...) are used to represent Russell (subset) expressions. c will be reserved for Boolean expressions appearing in loops and conditionals.

$M[\![a]\!]$ will be defined inductively on the structure of a. Thus normally we will define the meaning of expressions of a given operator nesting level from the meanings of expressions of lesser nesting levels. There will be one clause in the definition for each possible top level construct in the expression.

Sometimes it will be necessary to instead specify a collection of several possible meaning functions and then to state that the actual meaning function may be an

arbitrarily chosen member of that collection. For example, we have not completely specified the allocation functions A_T. This will give rise to several possible meanings for the NewT functions of which we may pick one.

We also use this mechanism to define the meaning of a non-terminating expression. Rather than introducing a special undefined element (usually written as ⊥) we will actually leave its meaning undefined by stating that essentially any function of the right type is acceptable as a meaning function for the construct.[6] We will then, in the following two chapters, construct proof rules which hold no matter which choice of meaning functions is made[7].

We abbreviate the state component of the meaning function, i.e.

$$(M[\![a]\!][e,s])_t$$

as $M_s[\![a]\!][e,s]$. Similarly we abbreviate the value component as $M_v[\![a]\!][e,s]$.

$M[\![a]\!]$ will be constructed so that it satisfies the four properties given below. We need the following definitions

2.2.1 Definition [a{} def]:

The notation $a\{x \leftarrow y\}$ denotes the Russell expression a with all occurrences of identifier x replaced by identifier y.[8] It is assumed that y does not occur in a. We will occasionally use this notation with lists of variables $\bar{x}$ and and $\bar{y}$. Thus $a\{\bar{x} \leftarrow \bar{y}\}$ denotes a with each x_i replaced by y_i.

---

[6]The motivation for this approach is given in chapter 6. It will not be apparent until we discuss the proof rules involved.

[7]All programs written in our Russell subset are deterministic. Thus we don't use this mechanism to model conventional nondeterminism. If we added such constructs this might be another appropriate use. See also the discussion in chapter 6.

[8]Since we are allowing only identifier for identifier substitution, it is safe to substitute for all occurrences of x, not just the "free" ones; we only have to guarantee that we substitute for the binding as well as the bound occurrences.

2.2.2 Definition [{}f def]:

We define for any function f:

$$(\{y \leftarrow v\}f)[x] = \begin{cases} f[x] & \text{if } y \neq x \\ v & \text{otherwise} \end{cases}$$

That is, $\{y \leftarrow v\}f$ is f modified so it returns v at y. We extend this definition to states s as follows:

$$\{y \leftarrow v\}s = (\{y \leftarrow v\}s_m, s_a)$$

The four previously mentioned properties are now stated as theorems. The proofs will also be by structural induction on the formulas. We give explicit proofs together with those clauses of the definition of M⟦a⟧ for which the inductive step is not obvious.

2.2.3 Theorem [e compactness thm]:

M⟦a⟧[e,s] = M⟦a⟧[e',s] whenever e and e' agree on those identifiers occurring free (i.e. outside the scope of a declaration for the identifier) in a. Less formally, let I_a be the set of identifiers occurring in a. Then the meaning of a depends only on the bindings of the identifiers in I.

2.2.4 Theorem [s compactness thm]:

The analog to the above for states is a bit harder to formalize. Informally we want to state that the meaning of a construct in two states depends only on the values of variables appearing in the construct and on whether or not they alias. Unfortunately this is not always true unless we are very careful in the definitions we introduce. (For example, NewT[] may produce different l-values in spite of the fact that there are no variable names mentioned in the expression.)

We say that a location $x \in L_T$ *has been allocated* in a state s if $s_a[T] \geq i$ where i is the natural number such that $A_T[i] = x$.

Consider a Russell expression a and a collection of identifiers I with var signature, such that all free identifiers in a with var signature are in I. We will compare the meaning the meaning of a in environment e and state s with that in e' and s'. Assume that for every $x \in I$ either both e[x] and e'[x] have been allocated, or they will be allocated at the same time, that is

$$e[x] - s_a[T] = e'[x] - s'_a[T]^{9}.$$

We say two states s and s' *match at locations* $x,y \in L_T$ (with respect to the environments e and e', and the set of identifiers I) if $s_m[x] = s'_m[y]$ and for every $z \in I$, $e[z] = x$ if and only if $e'[z] = y$. That is, two locations match if and only if the same value is stored there and they alias the same variables.

Define two state-value pairs (t, v) and (t', v') to be *similar* (relative to the initial states s and s', the initial environments e and e', and the set of identifiers I) if the following conditions hold. We assume v and v' are taken from D_S.

a) For all $x \in I$, t and t' match at e[x] and e'[x]. Further, for all types T

$$t_a[T] - s_a[T] = t'_a[T] - s'_a[T]$$

Less formally, t and t' are s and s' modified in essentially the same way.

---

[9] The restrictions on s_a are rather technical. We do not insist that identifiers be bound only to locations which have already been allocated. Thus an identifier free in a Russell expression a may alias a local variable allocated within a. We have to insure that such aliases occur in the same way in both states. Although this may appear confusing at the moment, its generality permits the very simple treatment of declarations in chapter 4.

b) If S is a val or func signature then

$$v = v'$$

c) If S is a func signature then consider the result obtained by (possibly repeated) application of the function value to arguments and states which match at any l-value arguments with respect to environments mapping (arbitrary) parameter names to the arguments (and the set of parameter names). (The latter part states that arguments with var signature must alias in the same way in all cases.) The result state and result value pairs for such applications must be similar (relative to the initial states and the same environments as above).

This will turn out to be a formal way of stating that applications of the function must themselves satisfy [s compactness thm].

d) If S is "var T", then s' and t' match at v and v'. If $v \neq e[x]$ for any $x \in I$ then (x is freshly allocated and thus)

$$v - s_a[T] = v' - t_a[T]$$

Assume states s and s' match at all pairs of locations e[x] and e'[x], for $x \in I$. Also assume that environments e and e' agree on bindings of identifiers outside I and that all functions in either environment satisfy condition (c) above, i.e. don't immediately contradict this theorem. Then the pairs ($M_s[\![a]\!][e,s]$, $M_v[\![a]\!][e,s]$) and ($M_s[\![a]\!][e',s']$, $M_v[\![a]\!][e',s']$) are similar.

It follows from the import rule that if a has func signature then then the condition on the states is vacuous. (This fact was already incorporated into clause (c) of the definition of similarity.) Note also that we do not insure that the values

at inaccessible locations are not changed; we only claim that they are not examined. In particular we view nonterminating computations as potentially modifying all locations, accessible or not.[10] The motivation for this will again become clear in chapter 4.

2.2.5 Theorem [s monotonicity thm]:

$M_s[\![a]\!][e,s]_a[T]$[11] is never less than $s_a[T]$. Less formally, evaluation of an expression never causes any locations to be deallocated.[11] Furthermore $M_v[\![a]\!][e,s]$ where a has signature var T, may produce only locations which are either bound to a free identifier in a, or are allocated in $M_s[\![a]\!][e,s]$, but not in s.

We again assume that e does not bind identifiers to functions whose (possibly repeated) application decreases $s_a[T]$ or returns unallocated, non-argument locations. Such bindings are never introduced by a Russell declaration. For the inductive proof to go through, we also argue that functions produced as the value component of the meaning of a Russell expression never violate this restriction.

2.2.6 Theorem [renaming thm]:

The value and state produced by an expression are not altered when an identifier is consistently renamed in both the environment and the expression. That is,

$$M[\![a]\!][e, s] = M[\![a\{x \leftarrow y\}]\!][\{y \leftarrow e[x]\}e, s]$$

[10]Terminating expressions do not modify other locations in a way that is visible to the rest of the program. They may however still modify locations allocated for local variables.

[11]This may be an undesirable property for a real implementation of the language. On the other hand even a real implementation should guarantee that the observable behavior is as if this property held.

Here y is an identifier not already occurring in a. Neither x nor y is a special constant identifier as described below.

We will write

$$(a,e) \sim (a',e')$$

when both

$$a' = a\{x \leftarrow y\}$$

for some x and y with either y not occurring in a or y = x, and e' agrees with

$$\{y \leftarrow e[x]\}e$$

on all the identifiers occurring in a'. Note that ~ is an equivalence relation. Using [e compactness thm], we can rewrite the theorem as

$$(a,e) \sim (a',e') \Rightarrow M[\![a]\!][e,s] = M[\![a']\!][e',s]$$

2.3. The Language

We now list the possible expressions in the language and inductively define their meaning. For each construct we give an informal definition, a formal inductive step in the definition of the meaning function, and, if necessary, the inductive steps in the proofs of the preceding four theorems.

The constructs to be considered initially are:

1. The identifier x. Its evaluation has no effect on the state. It yields the l- or r-value bound to x. Constants, such as the integer 17, are treated as special identifiers with fixed meanings.[12] Formally we have for the first case:

[12]For the sake of clarity we do not adopt the very general treatment of constants described in [Boe 80]. For example, we treat 1, 2, ... simply as constant identifiers denoting the corresponding integer values. In this context we are not concerned with the fact that there are infinitely many of them.

$$M[\![id]\!] = \lambda e \lambda s\,.\,(s, e[id])$$

In the constant case this becomes

$$M[\![id]\!] = \lambda e \lambda s\,.\,(s, c)$$

where c is the constant value associated with the identifier. In both cases M[id] trivially satisfies [s monotonicity thm], [s compactness thm], and [e compactness thm]. If we substitute for something other than the identifier in question [renaming thm] is trivially satisfied. If we substitute for the given identifier we only have to consider the non-constant case. Here we get

$$\begin{aligned} M[\![id\{id \leftarrow y\}]\!][\{y \leftarrow e[id]\}e, s] &= M[\![y]\!][\{y \leftarrow e[id]\}e, s] \\ &= (s, (\{y \leftarrow e[id]\}e)[y]) \\ &= (s, e[id]) \\ &= M[\![id]\!][e,s] \end{aligned}$$

2. The application of a simple built-in operator like integer +. Such operators are side effect free in that they have no effect on the state beyond that introduced by the arguments. For simplicity we consider only expressions with binary operators

$$a \text{ op } b$$

where op is +, * etc.

Argument expressions a and b are evaluated in that order. Op is applied to their respective values to yield the result.

Op can usually be applied only to r-values. The only exception we will consider is '=', which when applied to l-values yields true if and only if the two arguments are aliases. The result is without exception an r-value.

Each of these operations will have a function signature associated with it. The signature of such an expression is thus just the result part of this signature. We will insist that the signatures of the operand expressions and the parameter signatures are syntactically identical for the application to be legal.

We generally assume that these operations have **val** parameter signatures. The only operation considered initially which can be applied to l-values is an operator which yields true if and only if the two arguments alias. We will denote this operator, as well as the equality operators for various r-value domains as '='.

Formally we have

$$M[\![a \text{ op } b]\!] = \lambda e \lambda s \,.\, (M_s[\![b]\!][e,s'], \text{op}'[M_v[\![a]\!][e,s], M_v[\![b]\!][e,s']])$$

where s' abbreviates

$$M_s[\![a]\!][e,s]$$

and op' is the mathematical function corresponding to op, for example integer addition in the case of '+'.

We outline the induction step for [s compactness thm]. Let s and s' be two states which match at locations accessible in a or b with respect to environments e and e'. It follows from the induction hypotheses for [s monotonicity thm] and [s compactness thm] (in addition to the definition of similarity) that $M_s[\![a]\!][e,s]$ and $M_s[\![a]\!][e,s']$ will also match at these locations. We can then apply the induction hypothesis to $M_s[\![b]\!]$. If a and b have **val** signatures, this immediately gives the desired conclusion. In the case of equality of l-values, we know from the induction hypothesis for [s monotonicity thm] that a and b either return newly allocated values, or values bound in e or e' respectively. From the induction hypothesis for [s compactness thm] applied to the free identifiers in a and b (and the definition of matching locations), if either a or b does the latter in one state, then it must in both states. If either a or b returns a newly allocated

value then the result is always false and we're done. Otherwise they must both alias a free identifier. Since such aliasing is preserved, again the same result is produced in both state-environment pairs.

The proofs of [e compactness thm], [s monotonicity thm], and [renaming thm] are easy.

3. The expression

$$TV[a]$$

where T is a type and a must yield an l-value. It has no effect on the state. It yields the value stored at the location denoted by a. This corresponds to the ValueOf operation in Russell. In most other programming languages this operation is always implicit.

The expression a must have signature **var** T. TV[a] has signature **val** T. We usually abbreviate TV[a] as V[a] since the type T can be inferred from a. Formally we let

$$M[\![V[a]]\!] = \lambda e \lambda s \,.\, (s', s'_m[M_v[\![a]\!][e,s]])$$

where s' is $M_s[\![a]\!][e,s]$.

We prove the induction step for [s compactness thm] by first noting that the state component of the meaning is the same as that for the expression a. Thus the claims dealing with the state follow directly from the induction hypothesis. We also know from the induction hypothesis that given the two states s and s' which match at locations accessible in a, the resulting locations must match. Thus the same value must be stored there in both cases.

The induction steps for the other theorems are trivial.

4. The simple assignment

$$a := b$$

The expressions a and b must have signature **var** T and **val** T for the same T. Evaluation consists of evaluating a and b in that order. The value produced by b is then stored at the location specified by a. The value produced by a := b is that yielded by b. Likewise the signature is that of b.

Formally we let

$$M[\![a := b]\!] = \lambda e \lambda s \,.\, (\{M_v[\![a]\!][e,s] \leftarrow v\}(M_s[\![b]\!][e,s']),\ v)$$

where

$$s' = M_s[\![a]\!][e,s], \text{ and}$$
$$v = M_v[\![b]\!][e,s']$$

The induction step for [s compactness thm] is similar to that for a side effect free operation. We can use the same arguments to show that v and $M_s[\![b]\!][e,s']$ behave properly. Changing the state at $M_v[\![a]\!][e,s]$ maintains the similarity, since by the induction hypothesis the locations produced by a in different states must match. Certainly they will continue to match if we change the value stored there to the same fixed r-value produced by b.

5. The sequence

$$a; b$$

Evaluation possibly changes the state by evaluating a and b in succession. It yields the value of b as its value. Its signature is that of b.

Formally:

$$M[\![a; b]\!] = \lambda e \lambda s \,.\, M[\![b]\!][e,s']$$

where once more s' abbreviates

$$M_s[\![a]\!][e,s]$$

(This can be viewed as just another side effect free operation, namely a projection function. It is convenient to treat it specially in the logic.)

6. The conditional

if c **then** a **else** b **fi**

First c is evaluated. If it yields the value true then a is evaluated. Otherwise b is evaluated. The value produced is that of whichever of a or b was evaluated. The condition c must have signature **val** B. (We continue to use B for the Boolean type.) The signatures of a and b must be identical.[13] The signature of the conditional is that of a or b.

Formally let IF abbreviate the above construct. Then let

$$M[\![IF]\!][e,s] = M[\![a]\!][e,s'] \quad \text{if } M_v[\![c]\!][e,s] = \text{true}$$
$$= M[\![b]\!][e,s'] \quad \text{if } M_v[\![c]\!][e,s] = \text{false}$$

Here again

$$s' = M_s[\![c]\!][e,s]$$

We have tacitly assumed that all expressions are type correct, so that true, and false are the only values of c that need be considered.

[13] This restriction could be relaxed when the signature of the construct as a whole doesn't matter, for example if it appears as the first element of a sequence.

7. The loop

while c do a od

If c evaluates to **true**, a is evaluated and the process is repeated until c becomes **false**. The loop always yields the value **false**, the last value of c. It has signature val B.

To specify the meaning of the loop formally, we proceed as follows. We temporarily will consider extended "meaning functions" mapping $E \times S$ to P(S), the set of all collections of states. (The value component of the result will be dropped temporarily since it is not interesting. The notion of extended meaning functions will be refined considerably when we discuss recursive functions.) We can use such an extended meaning function to express the fact that a construct may result in several possible states. In particular we can explicitly leave the state resulting from a non-terminating loop undefined by saying that the extended meaning function for the loop produces the set of all possible states.

We now construct a sequence of extended meaning functions M_i for the loop which, when given $s \in S$ specifying some initial state and $e \in E$ will produce a set of possible final states after i iterations of the loop. Very informally, if we consider an "impatient" machine which gives up and produces a completely undefined result if the loop doesn't terminate after i or fewer iterations, then M_i is the extended meaning that such a machine would give to the loop. Thus $M_i[e,s]$ will be S if the loop does not terminate in i or fewer iterations, and the singleton set consisting of the final state if it does.

We will count the final evaluation of c as one iteration. Thus no loop ever terminates in zero iterations. Therefore

$$M_0 = \lambda e \lambda s . S$$

We then iteratively define M_{i+1} by

$$\begin{aligned} M_{i+1}[e,s] &= \{M_s[\![c]\!][e,s]\} && \text{if } M_v[\![c]\!][s,e] = \mathbf{false} \\ &= M_i[e, M_{body}[e,s]] && \text{if } M_v[\![c]\!][s,e] = \mathbf{true} \end{aligned}$$

where

$$M_{body}[e,s] = M_s[\![a]\!][e,M_s[\![c]\!][e,s]]$$

Note that if we define

$$M^0_{body}[e,s] = s$$

$$M^{i+1}_{body}[e,s] = M_{body}[e,M^i_{body}[e,s]]$$

then either there is a least $j < i$ such that $M_v[\![c]\!][e,M^j_{body}[e,s]]$ is **false** and $M_i[e,s]$ is $\{M_s[\![c]\!][e,M^j_{body}[e,s]]\}$, or otherwise $M_i[e,s]$ is S.[14] In the former case $M_k[e,s] = M_i[e,s]$ for all $k > i$.

We now define an extended meaning function M for the loop as a whole. It should produce a singleton set whenever any M_i does, and S otherwise. Formally we can write:

$$M[e,s] = \cap_i M_i[e,s]$$

We are assured by the previous observation that this intersection is non-empty.

(This construction is a very explicit version of the standard least fixpoint construction, as presented in [Sto 77] or [deB 80]. There is no reason to pursue it at a more abstract level here. When we look at recursive functions the

[14] We could have used this as the definition. The first version seems more elegant, though less useful.

equivalence of the two approaches will become even more apparent.)

This defines a function M for each loop with structural level corresponding to the current stage in the inductive definition. We explicitly indicate the dependence on the loop in question by writing M_{WH}. We can complete the construction, by taking m_{WH} to be any function mapping $E \times S$ to S satisfying

$$m_{WH}[e,s] \in M_{WH}[e,s]$$

and the following four constraints corresponding to the four theorems stated above.[15]

First, to insure the validity of [s monotonicity thm] we insist that m_{WH} be chosen so that if

$$m_{WH}[e,s] = s'$$

then $s_a[l] \leq s'_a[l]$. This is easy to arrange since there are no additional constraints on s'_a.

Second, m_{WH} should be chosen so that it satisfies [e compactness thm], that is, it should not differentiate environments which differ only on identifiers not occurring in the loop. This is possible since M_{WH} does not distinguish between such environments. (This in turn follows from the induction hypothesis.)

Third, we use the same approach to guarantee the validity of [s compactness thm].

[15]Our philosophy is that whatever treatment of nontermination is chosen is likely to be somewhat artificial, so we might as well arrange for no otherwise natural properties to be violated by nonterminating constructs. As we stated earlier, we may or may not want to consider [s monotonicity thm] as such a property.

Finally, we require

$$(WH,e) \sim (WH',e') \Rightarrow m_{WH}[e,s] = m_{WH'}[e,s]$$

This insures the validity of [renaming thm]. Once more M_{WH} obeys the same constraint, and thus this property is also easily satisfiable.

We finally define

$$M[\![WH]\!] = \lambda e \lambda s\ .\ (m[e,s], \mathbf{false})$$

For a nonterminating loop we have chosen an arbitrary final state (subject only to the constraint that it not violate any of the four properties stated at the beginning). Again our final proof rules will not allow us to conclude anything about that value, since we insist that they hold independently of the choice.

Consider however

```
while true do
   ...
od;
x := 1
```

All allowable meanings of this composite construct will produce final states with $x = 1$, and our rules will allow us to prove this.

8. The expression

NewT[]

for some type T. NewT yields a new l-value. The signature of NewT[] is **var** T.

For some of the formal arguments about our logic it will be useful to generalize the NewT functions so that they take an argument. Informally NewT[i], where i is some positive integer returns the i^{th} available location for objects of type T.[16] Formally we define the meaning of NewT in terms of the allocation function A_T which we previously introduced:

$$M[\![NewT[i]]\!] = \lambda e \lambda s\,.\,((\{n \leftarrow c_T\}s_m, \{T \leftarrow n\}s_a), A_T[n])$$

where n is $s_a[T] + M_v[\![i]\!][e,s]$[17] and c_T is a fixed constant of type T which is used to initialize new variables. (For integers it might be 0. Or we may introduce a special undefined value for that purpose.)

NewT[] is taken to be an abbreviation of NewT[1].

Note that NewT must initialize the location it produces in order for [s compactness thm] to hold. Also note that if NewT[i] can possibly return a location already bound to an identifier, then we have to insist, as we did, that $s_a[T]$ be fixed, so that it does so in all permissible cases.

9. The block

let x == a in b nl

Its effect on the state is that of first evaluating a and then evaluating b with x bound to the value produced by a. Note that a is evaluated in the old environment. (See letrec below however.) The block yields the value produced by b.

---

[16]This is admittedly contrived, especially since there is no observable difference in program behavior if we change the value of the argument. It is not absolutely necessary to introduce this construct at all, since we can (approximately) simulate the effect of NewT[i] with the loop

j := i; **while** j > 0 **do** NewT[]; j := j - 1 **od**

Nonetheless it seems to be the cleanest way of giving a development for which we can prove the relative completeness result of chapter 5. More elegant treatments lacking these formal properties will be suggested in chapter 4.

[17]Potential side effects of i are ignored. This is not unrealistic since an implementation could produce correct I/O behavior without evaluating i. In any case we will never use the construct in contexts in which i has side effects.

Formally we abbreviate the above bloci by LET and then write

$$M[\![LET]\!] = \lambda e \lambda s\,.\,M[\![b]\!][\{x \leftarrow M_v[\![a]\!]\}e, s']$$

where, once again,

$$s' = M_s[\![a]\!][e, s]$$

It is easily shown that this preserves [s monotonicity thm]. For [e compactness thm] and [s compactness thm] we note that although x is free in b and not in LET, b is evaluated in an environment in which x is bound to a value which is unaffected by the appropriate changes in the state or environment. Thus we can apply the induction hypothesis to this extended environment. In the case of [s compactness thm] with a **var** identifier x, we need to read unaffected as "matching in the two states".

For [renaming thm] we first apply the induction hypothesis to a and e to show that s' is not affected by the renaming. We then consider two cases. If x and y are the same identifier we have

$$\begin{aligned} M[\![LET\{x \leftarrow z\}]\!][\{z \leftarrow e[x]\}e, s] &= M[\![b\{x \leftarrow z\}]\!][\{z \leftarrow M_v[\![a]\!]\}\{z \leftarrow e[x]\}e, s'] \\ &= M[\![b\{x \leftarrow z\}]\!][\{z \leftarrow M_v[\![a]\!]\}e, s'] \\ &= M[\![b\{x \leftarrow z\}]\!][\{z \leftarrow e[x]\}\{x \leftarrow M_v[\![a]\!]\}e, s'] \\ &= M[\![b]\!][\{x \leftarrow M_v[\![a]\!]\}e, s'] \\ &= M[\![LET]\!][e,s] \end{aligned}$$

where the second to the last step follows from the induction hypothesis.

If we rename an identifier y other than x, then

$$M[\![LET\{y \leftarrow z\}]\!](\{z \leftarrow e[y]\}e, s] = M[\![b\{y \leftarrow z\}]\!][\{x \leftarrow M_v[\![a]\!]\}\{z \leftarrow e[y]\}e, s']$$
$$= M[\![b\{y \leftarrow z\}]\!][\{z \leftarrow e[y]\}\{x \leftarrow M_v[\![a]\!]\}e, s']$$
$$= M[\![b]\!][\{x \leftarrow M_v[\![a]\!]\}e, s']$$
$$= M[\![LET]\!][e,s]$$

and the conclusion follows from the induction hypothesis applied to b and $\{x \leftarrow M_v[\![a]\!]\}e$.

10. The function construction

$$\mathrm{func}\ [x_1: sig_1; \ldots; x_n: sig_n]sig_0\ \{\ a\ \}$$

$x_1, \ldots, x_n$ are the parameters to the function. a must have signature

sig_0. The signature of the construction is[18]

$$\mathrm{func}\ [sig_1; \ldots; sig_n]sig_0$$

We omit the signature information from function constructions when convenient.

The construct has no effect on the state. It yields the function that maps a given sequence of arguments to the value produced by a in the environment in which the parameter identifiers are bound to the arguments. Similarly the side effects produced by applying the resulting function are those of evaluating a in this environment.

Formally we abbreviate the above construction as FUNC and let

$$Q = T_{sig_1} \times \ldots \times T_{sig_n}$$

[18]Here we do not consider parameter names to be part of the signature. This avoids, in this restricted case, the renaming rule of [Boe 80].

We model the function as an element of

$$S \times Q \rightarrow S \times T_{sig}$$

More precisely, we let

$$M[\![FUNC]\!] = \lambda e \lambda s\, .\, (s, \lambda s \lambda \bar{y}\, .\, M[\![a]\!][\{\bar{x} \leftarrow \bar{y}\}e, s])$$

We again use the standard notation $\bar{x}$ as an abbreviation for $x_1, \ldots, x_n$.

It is easily shown that this definition is consistent with [e compactness thm], [renaming thm], and [s monotonicity thm]. Furthermore we note that the function f defined by $M_v[\![FUNC]\!][e,s]$ is itself monotone in s in the sense we require. That is, the allocation component of the state component of $f[s,\bar{y}]$ is pointwise greater than s_a.

For [s compactness thm] we must show that if the function is applied to states in which the arguments match, then the final state-result pairs are similar. That proof follows directly from the induction hypothesis applied to the body a and the environment in which only parameters are bound to l-values, and the fact that the only free variables in a are the parameters.

11. The function application

$$f[\bar{a}]$$

$\bar{a}$ is a sequence of comma separated expressions, $a_1, \ldots, a_n$. The effect on the state is that produced by first evaluating the expressions appearing in $\bar{a}$ and then applying the function with the results of the first evaluation used as arguments. The result produced is that given by the actual application of the function. The signature of the application is the result signature of f.

More formally, for a given environment e and state s we let v_i be the value produced by the i^{th} argument and s_i be the state after the evaluation of the i^{th} argument. Inductively,

$$s_0 = s$$
$$s_{i+1} = M_s[\![a_{i+1}]\!][e, s_i]$$
$$v_{i+1} = M_v[\![a_{i+1}]\!][e, s_i]$$

The reader should remember that the evaluation of f can have no side effects (as a consequence of the import rule).[19] We can now define the meaning of the application:[20]

$$M[\![f[\bar{a}]]\!][e,s] = (M_v[\![f]\!][e,s])[s_n,\bar{v}]$$

This preserves [s monotonicity thm], since by induction hypothesis $M_v[\![f]\!][e,s]$ has the desired property.

The argument for [s compactness thm] is a little more involved:

By the induction hypothesis applied to f, the application has the right similarity property with respect to environments in which (arbitrary) identifiers are bound to the arguments. From the induction hypothesis applied to the arguments we

---

[19]We are a bit sloppy here. It is possible that the evaluation of f affects some components of the state. But these components are not accessible by the program after the evaluation of f. Our formal semantics pretend that such effects are undone once f is evaluated. This is formally convenient, but imposes no constraint on an implementation since it cannot possibly affect the value of any expression with val signature. (It can affect the value produced by NewT, but the value returned will be indistinguishable from the one that would have been returned without the intervening evaluation of f.)

Nontermination of f will be modeled by allowing any conceivable function as the meaning for f. Thus the application of f will result in a completely undefined state even if we ignore the effect of the evaluation of f.

[20]Note that the meaning of the application itself does not depend on the environment at the time of the evaluation since we are using static binding. Thus the obvious inductive arguments go through for both [e compactness thm] and [renaming thm].

know that they match with respect to the real environments we're considering. By the same reasoning we used for the equality operation on l-values, this implies that aliasing among the arguments is preserved. Thus they also match with respect to the parameter-argument binding environments. It follows that the results are identical (in the case of a non-**var** signature) or match with respect to the parameter-argument environments. In the former case this is the only property we need of the value component. We consider the latter case further.

We need to show that an l-value result of the application consistently aliases the same identifiers in the environment (under the assumptions of [s compactness thm]). We argue that if it aliases an identifier x in one environment, then it must consistently do so. If it also aliases an argument then we're done since it has to alias that argument consistently, and the arguments match. If it doesn't alias any argument but aliases x, we know by [s compactness thm] that x was not allocated. Thus if x has signature **var** T we can assume that $e[x] - s_a[T]$ is fixed. The induction hypothesis tells us that the offset of the result from $s_a[T]$ is fixed. Thus the difference between x and the result must also be fixed. Thus they must alias consistently.

12. The recursive declaration

```
letrec
   x1 == a1
   x2 == a2
   .
   .
   .
   xn == an
in
   b
ni
```

which we abbreviate as LETREC.

The a_i all have function signature. Thus only b may have side effects on the state. The meaning of this is similar to a simple let-block, except that the a_i are evaluated in the new scope, and the x_i may appear on the right side of the declarations. Thus groups of mutually recursive functions may be declared.

For the sake of concise notation we will sometimes write the above construct as

letrec $\bar{x}$ == $\bar{a}$ **in** b **ni**

To deal with this construct formally we need to reintroduce extended meaning functions. This time we start with the set of all possible bindings for the x_i and then iteratively use these to define more restricted sets of possible bindings. Unfortunately we now have to define the "possible values" produced by one of the a_i given a set of bindings for the x_i. This is slightly more complicated than it appears since we need to distinguish between, for example, the set of all possible functions, and a single function producing all possible results. The former corresponds to a divergent expression which yields a function. It will be the extended meaning of f declared by


```
letrec
   f == f
in
   f
ni
```

The latter represents a function which diverges only when applied. It might be the meaning of f in

```
letrec
   f == func[x: val Integer] val Integer { f[x] }
in
   f
ni
```

The fact that one of the above blocks terminates and the other doesn't should make it clear that we need to introduce such distinctions.

We extend our notion of meaning function to interpret the a_i. We first introduce some extended domains on which these will be defined. The idea is to extend the domains we have previously defined with values representing nontermination. To be consistent with the remainder of our treatment we view these extended domains as containing sets of "possible" values. Thus the extended domain D_{Integer} will consist of all singleton sets of Integers, each of which may be the value of a terminating integer expression, and the set of all integers, which will be used as the value of a non-terminating integer expression.[21] For

[21] In essence we give a conventional least fixed point semantics based on complete partial orders. The addition of the whole set as a new element corresponds to the addition of $\perp$ in a conventional treatment. In the case of the loop we could pretend that we were working with the whole powerset rather than just singletons and the universe. We then showed that none of the intermediate sets arose. Here we explicitly have to avoid these intermediate sets, as is illustrated by the following:

Consider the block

letrec f == **func**[x: **val** *Rational*] { f[x]/2 } **in** f[1] **ni**

We take the type *Rational* to be all rational numbers in the interval (0,1]. If we proceed as with the loop, we might let the value component of the i^{th} approximation to the extended meaning of the block be the set $(0,2^{-i}]$, since these are the possible values if we expand i recurany set s we define

$$p(s) = \{\{x\} \mid x \in s\} \cup \{s\}$$

We let

$D_{\mathrm{val}\,T} = p(D_{\mathrm{val}\,T})$

$D_{\mathrm{var}\,T} = p(D_{\mathrm{var}\,T})$

$D_{\mathrm{func}[s_1, \ldots, s_n]\, s_0} = p((p(S) \times D_{s_1} \times \ldots \times D_{s_n}) \to D_{s_0} \times p(S))$

As before we define D to be the union of these sets.

We define the set of extended environments E to be the set of functions mapping identifiers to D. The set of extended states S is just p(S).

We say that a particular value v in D_t is more defined than a value V in D_t ($v \geq V$) if either $v \subseteq V$ or t is a function signature, v and V are both singletons, and for every extended state s and arguments $\bar{x}$, $v[s,\bar{x}] \geq V[s,\bar{x}]$.

We say that a value v in D_t is fully defined if it is a singleton, and, if t is a function signature, then if s is some state and $\bar{x}$ some sequence of values, then $v[s,\bar{x}]$ is fully defined whenever s and the x_i are singletons. Note the obvious 1-to-1 correspondence between the fully defined values of D_t and values in D_t.

We have to define extended meaning functions for various constructs in the language. These will map $E \times S$ into $D_t \times S$ where t is the signature of the construct. We denote the extended meaning of a construct b by E[[b]].

We define the extended meaning functions as the normal meaning functions, except that the definitions need to be adjusted as follows:

sive applications of f. We would then define the (value component of the) extended meaning of the block to be the intersection of these sets. The meaning of the block would be any element of this intersection. The problem of course is that the intersection is empty.

1. All recursive references to meaning functions are replaced by references to extended meaning functions. When necessary we apply the earlier construction to each element of a set, for example, to each an element of an extended state. Thus the extended meaning of the V operation is

$$\mathbf{E}[\![TV[a]]\!] = \lambda e \lambda s\,.\,(s', v)$$

where

$$s' = \mathbf{E}_s[\![a]\!][e,s]$$

and v is the set of all values v in R_T such that

$$v = t_m[l]$$

$$t \in s'$$

$$l \in \mathbf{E}_v[\![a]\!][e,s]$$

This and other definitions are modified as follows:

2. All meaning functions are strict in the state. That is,

$$\mathbf{E}[\![a]\!][e,S] = (D_t, S)$$

where t is the signature of a. Undefined states are mapped to undefined value-state pairs[22].

3. If the meaning of either argument to a builtin operation or an assignment is undefined then its result is undefined, i.e. it produces (D_t, S). The same applies to the condition in a conditional expression (but the unevaluated

[22] In contrast, note that E[[id]][e,s] where id is undefined in e does not yield an undefined state. We in fact assume that the binding of an identifier is not looked up unless it is needed. In particular we will treat

letrec f == f in f; 1 ni

as terminating. Given a reasonably intelligent compiler, this is not unrealistic; there is never a need to compute f.

branch may of course be undefined without producing an undefined result).

4. If the meaning of the operator or any of the arguments in a function application is not a singleton, then the result of the application is undefined. Note that the operator may produce an undefined value even if neither of these is true.

5. Formally the extended meaning of a function construction is defined like the regular meaning function (except that the meaning is made strict in the state). Thus function constructions may now produce singleton sets containing a function which, when applied, will produce an undefined result.

6. The extended meanings of loops and recursive declarations are derived from their normal meaning functions as for the other constructs. We define a sequence of extended meanings of the loop (or the right hand sides of a recursive declaration) in the extended environment and state under consideration. We then use the limit of this sequence directly to define the meaning of the construct, rather than making an arbitrary choice of a standard meaning function consistent with it[23].

---

[23]This treatment is inconsistent with the one for the loop in a subtle way. The semantics for the loop insure that the extended meaning of

while V[x] **do** β; x := **false od**

is a singleton, whether or not β terminates. For the recursive declaration we insist that all subexpressions terminate before we view the whole construct as terminating. It is easy to modify the semantics of the loop to be consistent with the recursive declaration. Alternatively we can modify the recursive declaration semantics to be consistent with the loop. (The latter involves defining extended meanings for loops and recursive declarations as we defined regular meanings, except that the domains $\mathcal{D}_t$ and $\mathcal{S}$ take the place of D_t and S, and that extended meaning functions replace ordinary meaning functions. Thus in giving the extended meaning of recursive declarations we will be dealing with values in, say, $p(p(D_t))$.)

The reason we do neither of these is that these distinctions have no practical significance, and the slightly inconsistent semantics lead to the simplest proof rules.

We are now ready to return to the meaning of recursive declarations. We successively approximate the objects that the x_i will be bound to. We will denote these objects by the list $\bar{y}$. We let $\bar{y}^i$ denote the extended meanings when only i levels of recursion are considered, that is when any recursive call at level > i is assumed to produce an undefined result. Thus we take y_j^0 to be the undefined element in $\mathcal{D}_{t_i}$ where t_i is the signature of a_i.

We then inductively let

$$\bar{y}^{i+1} = \mathbf{E}_v[\![a_i]\!][\{\bar{x} \leftarrow \bar{y}^i\}e, s]$$

where e and s are the fully defined extended environment and state corresponding to e and s. Note that $y_j^{i+1} \geq y_j^i$, that is the $\bar{y}^i$ form a component-wise ascending chain. (This is easily shown by induction. Clearly $y_j^1 \geq y_j^0$. Our definitions of extended meaning functions are monotone in the $\geq$ relation. If $y_j^i \geq y_j^{i-1}$, then $y_j^{i+1} \geq y_j^i$.)

Let v^i be any such ascending chain (i.e. $v^{i+1} \geq v^i$) in $\mathcal{D}_t$. We can define the limit v of the sequence as follows. Note that if there is an i such that v^i is a singleton then v^j for all j > i must be singletons. If there is no singleton in the sequence take v to be the appropriate undefined element. If there is a singleton and t is a **val** or **var** signature then all singletons appearing in the sequence must be identical. Take the singleton as the limit. If there is a singleton and t is a function signature then let v be the function which produces for each state s and arguments $\bar{x}$ the limit of the sequence $v^i[s,\bar{x}]$.

Thus we can take $\bar{y}$ to be the (component-wise) limit of the $\bar{y}^i$. Thus we may let

$$(s',v') = E[\![b]\!](\{\bar{x} \leftarrow \bar{y}\}e,s]$$

and then chose s' to be a simple (non-extended) state and v' to be a simple value, such that the corresponding fully defined extended states and values are more defined then s' and v'. We finally let

$$M[\![LETREC]\!][e,s] = (s',v')$$

S' and v' must again be chosen so [s monotonicity thm], [s compactness thm], [e compactness thm], and [renaming thm] not be violated. (To show that this is possible, we show by induction that extended meaning functions preserve the corresponding properties. These proofs are identical to the normal case.)

2.4. Summary of the language

We summarize the meaning of the constructs in the Russell subset. We will use s^a to abbreviate

$$M_s[\![a]\!][e,s]$$

The reader should refer to the preceding sections for the meaning of recursive declarations and function calls.

1a. $M[\![id]\!] = \lambda e \lambda s\,.\,(s, e[id])$

1b. $M[\![const]\!] = \lambda e \lambda s\,.\,(s, c)$

2. $M[\![a \text{ op } b]\!] = \lambda e \lambda s\,.\,(M_s[\![b]\!][e,s^a], op'[M_v[\![a]\!][e,s], M_v[\![b]\!][e,s^a]]$

3. $M[\![TV[a]]\!] = \lambda e \lambda s\,.\,(s^a, s^a_m[M_v[\![a]\!][e,s]])$

4. $M[\![a := b]\!] = \lambda e \lambda s\,.\,(\{M_v[\![a]\!][e,s] \leftarrow M_v[\![b]\!][e,s^a]\}(M_s[\![b]\!][e,s^a]), M_v[\![b]\!][e,s^a])$

5. $M[\![a; b]\!] = \lambda e \lambda s\,.\,M[\![b]\!][e,s^a]$

6. $M[\![\text{if } c \text{ then } a \text{ else } b \text{ fi}]\!][e, s] = M[\![a]\!][e,s^c]$ if $M_v[\![c]\!][e,s] = \text{true}$

$= M[\![b]\!][e,s^c]$ if $M_v[\![c]\!][e,s] = \text{false}$

7. $M[\![\text{while } c \text{ do } a \text{ od}]\!]$ defined above.

8. $M[\![NewT[i]]\!] = \lambda e \lambda s\,.\,((\{n \leftarrow c_T\}s_m, \{T \leftarrow n\}s_a), A_T[n])$

where n is $s_a[T] + M_v[\![i]\!][e,s]$

9. $M[\![\text{let } x == a \text{ in } b \text{ ni}]\!] = \lambda e \lambda s\,.\,M[\![b]\!][\{x \leftarrow M_v[\![a]\!]\}e, s^a]$

10. $M[\![func[\bar{x}]\ \{a\}]\!] = \lambda e \lambda s\,.\,(s, \lambda s \lambda \bar{y}\,.\,M[\![a]\!][\{\bar{x} \leftarrow \bar{y}\}e, s])$

11. $M[\![f[\bar{a}]]\!][e,s] = (M_v[\![f]\!][e,s])[s_n,\bar{v}]$ where $\bar{v}$ and s_n are defined above.

12. $M[\![\text{letrec } \bar{x} == \bar{a} \text{ in } b \text{ ni}]\!]$ defined above.

CHAPTER 3

THE LOGICAL FRAMEWORK

Several problems arise in an attempt to axiomatize the Russell subset described in the previous chapter using a Hoare style formalism. The most obvious problem is that we need some way to talk about the value produced by an arbitrary program segment. This can be done as in [Cun 76], [Kow 77], [Pri 77], and [Schw 78] by explicitly introducing one or more symbols to denote such values. A less obvious problem is that we can no longer identify logical formulas with Boolean programming language expressions, as in [Hoa 69] and most subsequent work on the subject, unless we are much more careful about the semantics of logical formulas.

Consider the "formula":

(1) $(x := 3; \text{true})$ and $V[x] = 3$

Does $V[x]$ refer to the value stored at x before or after the assignment $x := 3$?

This problem has been solved in the past by restricting assertions to contain only side effect free formulas. This is overly restrictive for our purposes.

We take a different approach which not only avoids this problem, but leads us to a logic which is generally much better suited to the language at hand.

One way to view our approach is that we directly specify the meaning of formulas such as (1). The primitive underlying our formalism does nothing but make the range of state changes explicit. The resulting logic is sufficient to define a construct similar to Dijkstra's weakest preconditions ([Dij 76] or [Gri 81]). Thus nothing else is needed. In this sense the logic is similar to that of [Con 77].

3.1. Base Logic Syntax We start with a largely standard first order logic BL which can be used to reason about the underlying domain D. We will not give any more detail about its structure or the basic function symbols we are given to express facts about it. Likewise we will later ignore the axioms and inference rules used to reason about D.

Since D already includes components which model Russell functions we will require that BL be suited to reasoning about at least equality of such functions. We will explicitly provide rules for reasoning about other aspects of these functions.

In the next chapter we present inference rules which make explicit reference to well-founded partial orders. Thus we require some facility to state and prove properties of relations as well.[1] It is not unreasonable to picture BL as a version of ZF set theory.[2]

Since l-values are embedded in D we should also be able to reason about their equality in BL[3].

We assume that we know how to axiomatize BL, since it is essentially a standard mathematical theory. We do however specify the syntax of a first order formula and specify its meaning in a model-theoretic sense. This is useful since it easily generalizes to the full logic.

---

[1]Here we mean the usual mathematical notion of relation. Note that this gives rise to the mathematical notion of a function, which is slightly different from that of a Russell function. In particular, we may conclude that mathematical functions are equal if they produce the same result for all possible arguments. Since Russell functions may potentially differ in how they modify the state, this is not true for them.

[2]We could presumably embed the versions of these concepts that are actually needed into a much smaller theory. This would be an interesting formal exercise, but we are not concerned with such issues here.

[3]We need to introduce another function which produces l-values in the next chapter. We will explicitly axiomatize its behaviour there.

We outline the definition of a formula:

3.1.1 Definition [term def]:

1. If x is a variable then x is a term. The constants **true** and **false** are terms, as are any other constants of R that we may chose to provide. These constants may have functional signatures. Thus we will view any basic function symbols (e.g. + for integer addition) as a special case of such constants. As described below, basic predicates are also in this category. We insist that all such constants have Russell style signatures associated with them. We assume that signatures of functional constants provided by the logic are state independent, that is none of the parameters have **var** signatures.[4]

2. If t_i are terms and t_0 has a function signature, then $t_0[t_1, ..., t_n]$ is a term. We insist that the signatures of $t_t, ..., t_n$ match the parameter signatures of t_0. The signature of the whole term is the result signature of t_0 (as for applications in Russell expressions). Note that we view basic predicates of BL to be a special case of this construction: Predicates are functions with result signature val B, i.e functions which return a Boolean value.

3. If t_1 and t_2 are terms with identical signatures then $t_1 = t_2$ is a term with signature **val** B. Note that we will write t_1 iff t_2 as $t_1 = t_2$.

4. If t_1 and t_2 are terms with signature val B then

[4]Equivalently we could model them as mathematical rather than Russell functions. Since we have not discussed the precise treatment of relations, we can be somewhat more precise with our present approach.

$$t_t \wedge t_2$$

$$t_t \vee t_2$$

$$\neg t_1$$

$$t_t \Rightarrow t_2$$

are terms with signature **val** B.[5]

5. If t is a term with signature **val** B then

$$\exists x{:}sig\ .\ t$$

$$\forall x{:}sig\ .\ t$$

are both terms with signature **val** B.

We take identifiers bound by quantifiers to be typed, or more precisely to have a signature. We will usually leave the signature to be inferred from context.

3.1.2 Definition [formula def]:

A formula is simply a term with signature **val** B. We will make the usual convention that any free variables in a formula are implicitly universally quantified. (Technically it should be clear from con ext what the signatures of the free variables are, so that the domain of the quantification is clear.)

3.2. Base Logic Semantics

We now give a formal definition of the meaning of a term and thus the validity of a formula. This definition is conventional, excepting the definition of quantification over signatures.

We assume we are given an interpretation I which assigns to each basic constant c a meaning $I_c \in D_{sig}$, where sig is the signature associated with c. We assume that

[5]Not all of these need be taken to be primitive. This doesn't matter here.

true and false are assigned their usual values. Thus we will feel free to confuse true and false as symbols with their interpretations. We will extend the interpretation I to give meaning I_t to any term t within some environment e which gives bindings for the free identifiers in t. Thus I is dependent on e. If we need to denote this explicitly we write I^e.

The clauses in the definition of I are intended to correspond to those of the definition of a formula:

3.2.1 Definition [I def]:

1. We are assuming we already have interpretations for constant symbols. The interpretation of a logical variable is just its binding in the environment e. Formally:

$$I^e_x = e[x]$$

2.

$$I_{f[t_1, \ldots, t_n]} = (I_f[s_0, I_{t_1}, \ldots, I_{t_n}])_1.$$

Note that the function is applied to some state s_0, as well as its specified arguments. We assume that logical functions behave like conventional mathematical functions, and therefore the choice of s_0 does not matter. We specify it only so we don't have to construct two distinct domains for logical and programming language functions.

We ignore the state produced by the function application.

3.

$$\begin{aligned} I_{t_1=t_2} &= \textbf{true} \quad \text{if } I_{t_t} = I_{t_2} \\ &= \textbf{false} \quad \text{otherwise} \end{aligned}$$

4.

$$\begin{aligned} I_{t_1 \wedge t_2} &= \textbf{true} \quad \text{if } I_{t_1} \text{ and } I_{t_2} \text{ are both true} \\ &= \textbf{false} \quad \text{otherwise} \end{aligned}$$

The definitions for $\vee$ and $\neg$ are similar.

5. We let E' be the set of all environments which agree with e for all identifiers except possibly x, and which map x to an element of D_{sig} such that the resulting environment does not violate the hypothesis of [s compactness thm]. We can then define

$$\begin{aligned} I_{\forall x:sig\,.\,t} &= \textbf{true} \quad \text{if for all } e' \in E' \; I^{e'}_t \text{ is true} \\ &= \textbf{false} \quad \text{otherwise} \end{aligned}$$

Similarly

$$\begin{aligned} I_{\exists x:sig\,.\,t} &= \textbf{true} \quad \text{if there is an } e' \in E' \text{ such that } I^{e'}_t \text{ is true} \\ &= \textbf{false} \quad \text{otherwise} \end{aligned}$$

The reader should note that the interpretation of a closed formula is independent of the environment e.

We now extend this notion of a formula by adding a new kind of term.

A Russell-term has the form

$$< \text{Russell\_expression} >$$

Intuitively, such a term represents the value of the expression within the brackets when executed starting in the current state. Identifiers inside Russell-terms can be bound either by declaration in a let or letrec block or by a quantifier preceding the term.

We then add a clause 0 to the definition of a formula:

0. If e is any Russell (subset) expression then <e> is a term.

If several Russell-terms appear in an expression, side effects produced by one of them never affect the others. Thus, the side effects produced by an expression in an assertion are confined to within that pair of brackets. (This convention will be extended later.) As an illustration, if we rewrite (1) as

$$< x := 3; \text{true} > \text{ and } < V[x] > \; = 3$$

then it is equivalent to <V[x]> = 3, and not to true.

Formally we define the interpretation of such a term with respect to environment e and state s as follows:

$$I^{e,s}_{<t>} \; = \; M_v[\![t]\!][e,s]$$

We need to make two modifications to the other clauses of [I def]. All interpretations of terms now depend on a state as well as an environment.

We say that a formula is valid iff its interpretation is true in all possible states.

Russell-terms may denote state dependent functions. There is no apparent way to identify these with functions in BL. Thus we explicitly restrict clause 2 of [formula def] to state independent operators.

Unlike Hoare logic, we can maintain a strict separation between logical and programming language operations. We never require that it be possible to substitute programming language expressions into logical ones. In this way we, in theory, gain some increased generality at the expense of explicitly axiomatizing all operations (even side effect free ones) in the programming language. For present purposes, there is no need to take advantage of this. Thus we do assume that the operations and constants provided by the logic are a superset of those provided by the programming language. This simplifies the axioms given in the next chapter.

The following definitions will help to establish the relationship between this logic and more conventional ones.

3.2.2 Definition [< > t def]:

We let <a> t be the term t with each Russell term <b> appearing in t syntactically replaced by <a; b>. (We assume that bound variables in t are first renamed to avoid clashes with a.) Thus

$$< x := 3 > \; (< V[x] > - < V[z] > + \; a)$$

is equivalent to

$$< x := 3; V[x] > \; - \; < x := 3; V[z] > \; + \; < x := 3; a >$$

Informally we think of <a> t as the value of t after executing a. Observe that if t is a formula <a> t in this notation is very similar to the same construct in dynamic logic (see e.g. [Har 79]), to the construct S; t in [Con 77], or to wp(S, t) in Dijkstra's notation [Dij 76]. [Mir 71] introduces a similar construct which also applies to general terms rather than just formulas.

We will use this notation frequently.

3.2.3 **Definition [{} def]**:

Hoare's notation {P} S {Q} will be used occasionally to represent the formula $P \Rightarrow \langle S \rangle Q$. Again this corresponds closely to the standard use of the notation.

We conclude this section with a theorem justifying our informal interpretation of $\langle\rangle t$. We prove:

3.2.4 **Theorem [< > t meaning thm]**:

$$I^{e,s}_{\langle a\rangle t} = I^{e,s'}_{t}$$

where

$$s' = M_s[\![a]\!][e,s]$$

Proof:

We proceed by induction on the structure of t.

0. If t is the Russell term $\langle b\rangle$ we have

$$\begin{aligned} I^{e,s}_{\langle a\rangle t} &= I^{e,s}_{\langle a;\, b\rangle} \\ &= M_v[\![a;\, b]\!][e, s] && \text{(by [I def])} \\ &= M_v[\![b]\!][e, s'] && \text{(meaning of ;)} \\ &= I^{e,s'}_{t} && \text{(by [I def])} \end{aligned}$$

1. For constants or variables t, $\langle a\rangle t$ is equivalent to t by [< > t def] and the interpretation is independent of the state. Thus the equality clearly holds.

2. If t has the form

$$f[a_t, \ldots, a_n]$$

We know by induction hypothesis that for each a_i

$$I^{e,s}_{\langle a\rangle a_i} = I^{e,s'}_{a_i}$$

and similarly

$$I^{e,s}_{\langle a\rangle f} = I^{e,s'}_{f}$$

(Actually we know that the state doesn't matter for the interpretation of f. But we don't need that property here.) Temporarily using I to denote $I^{e,s}$ and I' to denote $I^{e,s'}$, we have:

$$\begin{aligned} I_{\langle a\rangle\ f[a_t, \ldots, a_n]} &= I_{(\langle a\rangle f)[(\langle a\rangle a_1), \ldots, (\langle a\rangle a_n)]} \\ &= (I_{\langle a\rangle f}[s_0, I_{\langle a\rangle a_1}, \ldots, I_{\langle a\rangle a_n}])_1 \\ &= (I'_f[s_0, I'_{a_t}, \ldots, I'_{a_n}])_1 \\ &= I'_{f[a_1 \ldots, a_n]} \end{aligned}$$

3, 4. The proofs for equality and Boolean connectives are essentially identical to case 2 above.

5. The term t may have the form

$$\forall x\colon sig\ .\ t'$$

or

$$\exists x\colon sig\ .\ t'$$

We consider the former case.

By [< > t def] we have

$$<a> \forall x: sig\ .\ t' \quad = \quad \forall x: sig\ .\ <a> t'$$

Again we know from the induction hypothesis that for all e

$$I^{e,s}_{<a>t'} \;=\; I^{e,s'}_{t'}$$

If we define a set of environments E' as in [I def] we get

$$\begin{aligned} I^{e,s}_{<a>t} &= \forall e' \in E'\ .\ I^{e',s}_{<a>t'} = true \\ &= \forall e' \in E'\ .\ I^{e',s}_{<a>t'} = true \\ &= \forall e' \in E'\ .\ I^{e',s'}_{t'} = true \\ &= I^{e,s'}_{t} \end{aligned}$$

The case of the existential quantifier is identical.

3.3. The Meaning of Inference Rules

We will develop inference rules which allow us to prove formulas which are valid for all allowable assignments of meanings to the Russell terms.

3.3.1 **Definition [glbl validity def]:**

A formula is ***globally valid*** iff it is valid for all possible meaning functions consistent with the definitions in chapter 2.

In the following discussion we will build inference rules of the form

$$\begin{array}{l} P_1, \ldots, P_n \\ \vdash \\ Q \end{array}$$

These should be read as: "If P_1, ..., and P_n are globally valid formulas, then so is Q. P_i and Q are formulas with free term variables in them. The inference rule may be

applied with any collection of terms (satisfying any implicit type constraints) consistently substituted for the term variables.

Both the P_i and Q in the above rule are intended to be closed formulas. Any free variables introduced by the substitution for the term variables are assumed to be universally quantified *over each formula individually*.

We can give a more formal statement of the meaning of the above rule. Consider any proper substitution of terms for term variables in the rule. Let P'_i and Q' be P_i and Q with the appropriate substitutions performed. Let I be the collection of interpretations consistent with chapter 2. In the following let $I \in \mathbf{I}$. We say the above rule is valid if

$$\begin{array}{l} \forall I,e,s\ .\ I^{e,s}_{P'_1} \ \wedge \\ \ldots \\ \forall I,e,s\ .\ I^{e,s}_{P'_n} \\ \quad \Rightarrow \ \forall I,e,s\ .\ Q' \end{array}$$

The quantification over e is intended to be restricted to those environments that establish "type correct" bindings, that is, it is assumed that each identifier is bound to an element of the domain corresponding to its signature.

We will frequently state inference rules without premises. We call such rules axioms (or more correctly axiom schemas). We omit the $\vdash$ symbol in this case.

3.4. Consequence and Composition Theorems

We will not concern ourselves with axioms dealing purely with the predicate calculus, equality, or the domain D. A simple adaptation of a standard mathematical theory will do. We do require that whatever axiomatization we chose here will allow us to substitute equals for equals. Thus it should include a rule like the following:

[= subst]:

$$t_1 = t_2 \;\Rightarrow\; t\{x \leftarrow t_1\} = t\{x \leftarrow t_2\}$$

Here we use the notation $t\{t_i/x\}$ to represent the term t with all free occurrences of x replaced by t_i. We assume that free variables in t_i are renamed before the substitution to avoid captures by quantifiers in t, and that x does not appear inside a Russell term.

An immediate consequence of our definition of the validity of a formula is

[U validity rule]:

$$\begin{array}{l} P \\ \vdash \\ \langle a \rangle\; P \end{array}$$

This states that if P is valid, i.e. holds for all states, then evaluating a results in a state in which P holds. The formal soundness proof of this rule follows immediately from [< > t meaning thm].

Two rules of inference usually presen in a programming logic follow immediately from the predicate calculus axioms and [U validity rule]. They are Hoare's rules of consequence and composition.

3.4.1 Theorem [consequ thm]:

$$\{P\}\; a\; \{Q\},\; Q \Rightarrow R$$

$$\{P\}\; a\; \{R\}$$

Proof:

We can rewrite the assumptions as

$$\begin{array}{c} P \Rightarrow \langle a \rangle\; Q, \text{ and} \\ Q \Rightarrow R \end{array}$$

By [U validity rule] the second assumption gives rise to

$$\langle a \rangle\; (Q \Rightarrow R)$$

This is by [< > t def], the same as

$$(\langle a \rangle\; Q) \Rightarrow (\langle a \rangle\; R)$$

Combining the above with the first hypothesis yields

$$P \Rightarrow (\langle a \rangle\; R)$$

or

$$\{P\}\; a\; \{R\}$$

which was the desired conclusion. ●

We can now easily derive a version of Hoare's statement composition rule as well:

3.4.2 Theorem [comp thm]:

$$P \Rightarrow \langle a \rangle\; Q,\;\; Q \Rightarrow \langle a \rangle\; T$$

$$P \Rightarrow \langle a; b \rangle\; T$$

Proof:

This follows immediately if we let R be $\langle b \rangle$ T in the previous theorem. ●

The following sections will present axioms and inference rules for specific programming language constructs. Aside from our treatment of non-termination these will be generalizations of Hoare's rules.

CHAPTER 4

AN AXIOMATIZATION

What follows is a collection of axioms and inference rules for the Russell subset defined in chapter 2. Unlike the rule given in the preceding chapter these explicitly mention programming language constructs. Each rule will be justified first informally, and then formally by proving its validity with respect to the denotational semantics for the same language given in chapter 2.

In general each construct will have two rules associated with it, an effect rule and a value rule. The effect rule describes its effect on the state by defining the meaning of an arbitrary term evaluated in the state produced by the construct. The value rule describes the value yielded by that construct.

4.1. Identifiers

We start with an axiom defining the effect on the state of evaluating a non-constant identifier:

[id ef ax]:

$$\langle x \rangle\ t = t$$

where x is an identifier and t is any term.

This axiom expresses the fact that evaluation of an identifier does not modify the state of the computation, and thus the value of a subsequent expression is not affected by it.

Soundness Proof: We must show

$$I^{e,s}_{\langle x \rangle t = t} = \text{true}$$

or equivalently

$$I^{e,s}_{\langle x \rangle t} = I^{e,s}_{t}$$

We have by [< > t meaning thm]

$$I^{e,s}_{\langle x \rangle t} = I^{e,s'}_{t}$$

where $s' = M_s[\![x]\!][e,s]$. But by the meaning given to identifiers in chapter 2, $s' = s$.
●

It follows from [< > t def] that the following formulation of the axiom is valid as well:

$$\langle x; a \rangle = \langle a \rangle$$

It is actually equivalent. The equality $t = \langle x \rangle\ t$ can be derived by first deriving equalities of the corresponding terms in t and $\langle x \rangle\ t$ using the above axiom, and then substituting equals for equals. The first notation is more convenient to use, and thus will be used in stating the remaining axioms.

The preceding effect axiom has a very general corresponding value axiom:

[id val ax]:

$$\langle a; x \rangle = x$$

Here x is any identifier and e is any expression.

Remember that if x is a variable $\langle e; x \rangle$ denotes the location bound to x, which cannot be modified by assignments. It does not refer to the value stored at the location.

Soundness Proof:

$$\begin{aligned}
\Gamma^{e,s}_{<a;\, x>} &= M_v[\![a;\, x]\!][e, s] && \text{(by [I def])}\\
&= M_v[\![x]\!][e, s'] && \text{for appropriate } s' \text{ (meaning of ;)}\\
&= e[x] && \text{(meaning of identifier)}\\
&= \Gamma^{e,s}_{x} && \text{([I def] once more)} \quad \bullet
\end{aligned}$$

The same rules apply to constant identifiers as well, provided equivalent constants are provided by the logic, as we assumed in the last chapter. The soundness proofs are identical.

4.2. Built-In operators

Recall that our language uses left to right argument evaluation. The cumulative side effects of evaluating a op b are thus identical to those of evaluating a; b. We describe this formally by the effect axiom:

[op ef ax]:

$$<a \text{ op } b>\ t \;=\; <a;\, b>\ t$$

Again, we are defining the side effects of an expression by stating how the value of a subsequent term is affected.

Soundness Proof: Follows immediately from [< > t meaning thm] and from the fact that the state components of the meaning functions for a; b and a op b are identical.

●

We now need a rule describing the value computed by a op b. We might want an axiom of the form:

$$<a \text{ op } b> \;\stackrel{?}{=}\; <a> \text{ op } <b>$$

The above however does not hold if the value of b is changed by previously executing a. Thus we rewrite the value axiom as:

[op val ax]:

$$<a \text{ op } b> \;=\; <a> \text{ op } <a;\, b>$$

Soundness Proof:

$$\begin{aligned}
\Gamma^{e,s}_{<a \text{ op } b>} &= M_v[\![a \text{ op } b]\!]\\
&= op'\,[M_v[\![a]\!][e,s],\, M_v[\![b]\!][e,\, M_s[\![a]\!][e, s]]]\\
&= op'\,[\Gamma^{e,s}_{<a>},\, M_v[\![a;\, b]\!][e, s]]\\
&= op'\,[\Gamma^{e,s}_{<a>},\, \Gamma^{e,s}_{<a;\, b>}]\\
&= \Gamma^{e,s}_{op}\,[\Gamma^{e,s}_{<a>},\, \Gamma^{e,s}_{<a;\, b>}]\\
&= \Gamma^{e,s}_{<a>} \text{ op } <a;\, b> \quad \bullet
\end{aligned}$$

We turn to the builtin V operation. In general we cannot say anything about the value produced by it, unless the argument was previously assigned to. In that case the assignment axiom will allow us to reason about it. The only property we state here describes the behavior of V with a complex expression as an argument. It can be viewed as a special case of [arg ax][1] which will be given below in the context of arbitrary function applications.

[V arg ax]:

$$x = <a> \;\Rightarrow\; <a;\, V[x]> \;=\; <V[a]>$$

---

[1]We could also simplify [op val ax] by treating primitive operations as function calls (as is done in full Russell) and allowing the use of the general version of [arg ax] in that context.

Note that when a is just an identifier y we may use [id ef ax] to specialize this to

4.2.1 Theorem [V = thm]:

$$x = y \Rightarrow \langle V[x]\rangle = \langle V[y]\rangle$$

Another interesting consequence of [V arg ax] is

4.2.2 Theorem [V comp thm]:

$$\langle a;\, V[b]\rangle = \langle V[a;\, b]\rangle$$

Proof:

By [V arg ax]

$$x = \langle a;\, b\rangle \Rightarrow \langle a;\, b;\, V[x]\rangle = \langle V[a;\, b]\rangle$$

and also

$$\langle a\rangle\ (x = \langle b\rangle \Rightarrow \langle b;\, V[x]\rangle = \langle V[b]\rangle)$$

The latter can be rewritten as

$$x = \langle a;\, b\rangle \Rightarrow \langle a;\, b;\, V[x]\rangle = \langle a;\, V[b]\rangle$$

Combining this with the first statement gives

$$x = \langle a;\, b\rangle \Rightarrow \langle V[a;\, b]\rangle = \langle a;\, V[b]\rangle$$

Since x is universally quantified and does not occur in the consequence of the implication, we obtain the desired result. ●

Soundness Proof: We return to the soundness of [V arg ax]. Fix e and s. Assume that $x = \langle a\rangle$ holds in the given state and environment. This means

$$e[x] = \Gamma^{e,s}_{\langle a\rangle} = M_v[\![a]\!][e,s]$$

We let s' be $M_s[\![a]\!][e,s]$. Then

$$\begin{aligned}
\Gamma^{e,s}_{\langle V[a]\rangle} &= M_v[\![V[a]]\!][e,s] \\
&= s'_m[M_v[\![a]\!][e,s]] \\
&= s'_m[e[x]] \\
&= s'_m[M_v[\![x]\!][e,s']] \\
&= M_v[\![V[x]]\!][e,s'] \quad (\text{since } M_s[\![x]\!][e,s'] = s') \\
&= M_v[\![a;\, V[x]]\!][e,s] \\
&= \Gamma^{e,s}_{\langle a;\, V[x]\rangle}
\end{aligned}$$ ●

We add the obvious effect axiom:

[V ef ax]:

$$\langle V[x]\rangle\ E = \langle x\rangle\ E$$

Soundness Proof: Immediate from [<>t meaning thm] and the respective definitions of the meaning functions. ●

As an illustration of the use of these axioms consider the expression

$$\langle V[x] * V[y] + V[z]\rangle$$

Using the two preceding axioms we have

$$\begin{aligned}
\langle V[x] * V[y] + V[z]\rangle &= \langle V[x] * V[y]\rangle + \langle V[x] * V[y];\, V[z]\rangle \\
&= \langle V[x] * V[y]\rangle + \langle V[x];\, V[y];\, V[z]\rangle \\
&= \langle V[x]\rangle * \langle V[x];\, V[y]\rangle + \langle V[x];\, V[y];\, V[z]\rangle
\end{aligned}$$

Applying [V ef ax] a few times gives us

$$\langle V[x] * V[y] + V[z]\rangle = \langle V[x]\rangle * \langle V[y]\rangle + \langle V[z]\rangle$$

By convention we drop the angle brackets around applications of V to a side effect free expression. Thus we can write

$$\langle V[x] * V[y] + V[z]\rangle = V[x] * V[y] + V[z]$$

In general if all operators other than V in a programming language expression a behave like the canonical side effect free operation op above we can (and do) neglect to distinguish between $\langle a \rangle$ and a.

4.3. Assignment

We obtain an assignment axiom which is in some sense simpler than Hoare's, especially when we consider the complete generality of this approach. Our first axioms explicitly deal only with the value produced by the V function immediately after the assignment and with the value produced by a NewT call immediately after the assignment. Unlike Hoare's logic, our formalism is sufficiently general that we can then deduce statements about $\langle x := e \rangle$ E for more general E.

[:= ef ax]:

$$\langle a \rangle = x \;\Rightarrow\; \langle a := b;\, V[x] \rangle = \langle a;\, b \rangle$$

$$\langle a \rangle \text{ "}\neq\text{" } x \;\Rightarrow\; \langle a := b;\, V[x] \rangle = \langle a;\, b;\, V[x] \rangle$$

$$\langle a := b;\, NewT[k] \rangle = \langle a;\, b;\, NewT[k] \rangle$$

Here a is an expression with **var** signature, b has **val** signature, and x is an identifier. By $\langle a \rangle$ "$\neq$" x we mean $\langle a \rangle \neq x$ or a and x have different signatures.

The first part of the axiom can be read as: If a and x are aliases (or the same identifier), then the value stored at that location after the assignment is the value yielded by the expression on the right of the assignment operator. Note that we have to write this value as $\langle a;\, b \rangle$ and not as $\langle b \rangle$ since the left expression is evaluated first, and we are thus interested in the value of b *after* the evaluation of a.

The second part of the axiom states that if a and x don't alias then the effect of the assignment on the value stored at x is the same as just evaluating a and b in succession.

The reader should recall our assumption about partial aliases from chapter 2. We have (temporarily) ignored the possibility of partial overlap between $\langle a \rangle$ and $\langle x \rangle$.

Soundness Proof: Consider the case of two aliased variables first. Thus we fix an environment e and state s and let

$$s' = M_s[\![a]\!][e, s]$$

$$s'' = M_s[\![a := b]\!][e, s]$$

$$v = M_v[\![b]\!][e, s']]$$

We assume

$$M_v[\![a]\!][e, s] = e[x] = M_v[\![x]\!][e, s']$$

We then show that

$$M_v[\![a := b;\, V[x]]\!][e, s] = M_v[\![a;\, b]\!][e, s] = v$$

This again follows easily from the definition of the meaning functions:

$$\begin{aligned} M_v[\![a := b;\, V[x]]\!][e,s] &= M_v[\![V[x]]\!][e, s''] \\ &= s''[M_v[\![x]\!][e, s'']] \\ &= s''[e[x]] \\ &= (\{M_v[\![a]\!][e,s] \leftarrow v\}\, M_s[\![b]\!][e, s'])\, [e[x]] \\ &= v \end{aligned}$$

In the unaliased case we define s', s", and v as before and proceed similarly:

$$
\begin{aligned}
M_v\llbracket a := b;\ V[x]\rrbracket[e,s] &= s''[e[x]] \\
&= (\{M_v\llbracket a\rrbracket[e,s] \leftarrow v\}\ M_s\llbracket b\rrbracket[e, s'])\ [e[x]] \\
&= M_s\llbracket b\rrbracket[e, s']\ [e[x]] \\
&= M_s\llbracket a;\ b\rrbracket[e, s]\ [e[x]] \\
&= M_v\llbracket V[x]\rrbracket[e, M_s\llbracket a;\ b\rrbracket[e, s]] \\
&= M_v\llbracket a;\ b;\ V[x]\rrbracket[e, s]
\end{aligned}
$$

We finally consider the axiom dealing with the NewT function. We observe that s'' (again defined as before) is just $M_s\llbracket b\rrbracket[e, s']$ or $M_s\llbracket a; b\rrbracket[e, s]$ with one altered value of s_m. Since the location returned by NewT[k] depends only on s_a, the conclusion follows. ●

We have stated the first parts of [:= ef ax] for the value of a simple variable following the assignment. The first line can be easily generalized to arbitrary expressions for x. The only change is that the condition $\langle a\rangle = \langle x\rangle$ becomes $\langle a\rangle = \langle a := b; x\rangle$. This version easily follows from the given one: Assume we want to simplify $\langle a := b; V[d]\rangle$. We have

$$\forall y\ .\ \langle a\rangle = y \Rightarrow \langle a := b; V[y]\rangle = \langle a; b\rangle$$

(Making one of the implied quantifiers explicit.) If we now consider $y = \langle a := b; d\rangle$ we have

$$\langle a := b\rangle\ y = y = \langle a := b\rangle\ \langle d\rangle$$

and thus

$$\langle a := b\rangle\ (y = \langle d\rangle)$$

by [V val ax]

$$y = \langle d\rangle \Rightarrow \langle V[y]\rangle = \langle V[d]\rangle$$

and thus by [U validity rule] and the preceding statement

$$\langle a := b\rangle\ (V[y] = V[d])$$

Thus

$$
\begin{aligned}
&\forall y\ .\ \langle a\rangle = y \wedge y = \langle a := b; d\rangle &&\Rightarrow \langle a := b; V[d]\rangle = \langle a := b; V[y]\rangle \\
&\forall y\ .\ \langle a\rangle = y \wedge y = \langle a := b; d\rangle &&\Rightarrow \langle a := b; V[y]\rangle = \langle a; b\rangle \\
&\langle a\rangle = \langle a := b; d\rangle &&\Rightarrow \langle a := b; V[d]\rangle = \langle a; b\rangle
\end{aligned}
$$

Using the same approach we can show for the other case

$$y = \langle a := b; d\rangle \wedge y \neq \langle a\rangle \Rightarrow \langle a := b; V[d]\rangle = \langle a; b; V[y]\rangle$$

The example below will give some indicatton that these rules are adequate to simplify a general term $\langle a := b\rangle\ t$. This will be dealt with more formally in the next chapter.

As an illustration we again consider the formula

$$\langle x := 3\rangle\ (V[x] - V[z] + a)$$

We assume that x and z do not alias.

We previously pointed out that this is equivalent to

$$\langle x := 3; V[x]\rangle - \langle x := 3; V[z]\rangle + \langle x := 3; a\rangle$$

Using [id val ax] this can now be simplified to

$$\langle x := 3; V[x]\rangle - \langle x := 3; V[z]\rangle + a$$

Thus we can apply [:= ef rule] to rewrite the above as

$$\langle x; 3\rangle - \langle x; 3; V[z]\rangle + a$$

Applying [id ef ax] a few times this yields

$$3 - V[z] + a$$

Since this is somewhat tedious we will observe for future reference that if a and b have no side effects, e.g., if they contain only V, +, * etc. $<a := b>\ t$ can be rewritten as t with $<V[d]>$ replaced by $<b>$ if $<a>\ =\ <d>$. This is essentially Hoare's substitution rule.

The value axiom for assignment is trivial:

[:= val ax]:

$$<a := b> \;=\; <a;b>$$

Soundness Proof: The value components of the meaning functions for a := b and a; b are identical.

4.4. Sequences

There are no explicit proof rules or axioms defining the meaning of ';'. As the preceding sections indicate, we instead define the meaning of a; b by dealing with different expressions a separately. These are just the effect rules associated with each language construct.

Thus a canonical approach to simplifying $<a_1; \ldots; a_n>$ would be as follows. First simplify $<a_n>$ to an equivalent term t containing no Russell terms except those of the form $<V[a]>$. This will involve primarily the use of value axioms. Then use [U validity rule] to rewrite the original term as $<a_1; \ldots, a_{n-1}>\ t$. Now take $<a_{n-1}>t$, expanding it as necessary according to [$<>$t def] and then simplify it to the above form. This time this will involve primarily the application of effect rules. This last step can then be repeated for all the preceding a_i.

It is of course not always necessary to use this particular approach. If we are trying to prove

$$\{P\}\ a_1; \ldots; a_n\ \{Q\}$$

It is easy to justify formally the Hoare approach of annotating the program with assertions between two adjacent "statements". Recall we showed in the last chapter that Hoare's composition rule is valid in this system as a derived rule. Similarly Dijkstra's calculus of weakest preconditions [Dij 76] could be interpreted and justified within this system, provided we restricted ourselves to a deterministic version of his language. This will be discussed in more detail in chapter 6.

4.5. Conditionals

The proof rules for the conditional are obtained essentially by translating the well-known one into this formalism. This is made easier if we first introduce the following shorthand notation. It is sometimes easy to give an axiom for a construct in terms of a complete equivalence, that is to define it in terms of another construct which is equivalent both in terms of the value and the state it produces. We use ≡ to denote this strong kind of equivalence. Formally $<a> \equiv <b>$ iff for every term t

$$<a> = <b> \;\wedge\; <a>t = <b>t$$

Thus we may write

[X ax]:

$$<a> \;\equiv\; <b>$$

instead of

[X val ax]:

$$<a> \;=\; <b>$$

[X ef ax]:

$$\langle a\rangle t \;=\; \langle b\rangle t$$

If we abbreviate

if c **then** a **else** b **fi**

as IF, we now have

[if ax]:

$$\begin{array}{rcl} \langle c\rangle & \Rightarrow & \langle IF\rangle \equiv \langle c; a\rangle \\ !\,\langle c\rangle & \Rightarrow & \langle IF\rangle \equiv \langle c; b\rangle \end{array}$$

Soundness Proof: It follows immediately from the definitions of the meaning functions that if $M_v[\![c]\!][e, s]$ is true then

$$M[\![IF]\!] \;=\; M[\![c; a]\!]$$

Similarly in the other case

$$M[\![IF]\!] \;=\; M[\![c; b]\!]$$

The soundness of the above rules is now clear.

4.6. The Loop Rule

We could derive a **while** rule based on Hoare's. The result however would be more complex than necessary. The problem is that the standard statement of the rule involves a hypothesis of the form

$$P \Rightarrow \langle a\rangle\ P$$

which states that the invariant is preserved when the body of the loop is executed. However, one of the things one might wish to prove in our present logic is that the value of some, say integer, term t is unaffected by the execution of the loop. That is,

$$\langle \textbf{while} \ldots \textbf{od}\rangle\ t \;=\; t$$

This requires some notion of a non-Boolean invariant. The obvious solution is to generalize the above hypothesis to arbitrary, not necessarily Boolean, P. This however leaves us with the question of what to do with the '$\Rightarrow$' operator in the hypothesis, whose arguments must be Boolean values.

There are several possible solutions. The one we chose involves generalizing the implication to an arbitrary partial order $\subset$. This makes our 'invariants' sufficiently general that we no longer need a separate notion of a variant function to show termination. Instead we just insist that $\subset$ is well founded and that what used to be our 'invariant' is in fact strictly decreasing with each loop iteration. Thus we obtain the following, at first glance strange, formulation. If c has no side effects and we use WH to denote

while c **do** a **od**

then the following rule holds:

$$\begin{array}{l} \subset \text{ is a well-founded partial order,} \\ \langle c\rangle \text{ and } v \subset k \;\Rightarrow\; (\langle a\rangle\ v) \subset v \\ \vdash \\ v \subset k \;\Rightarrow\; (\langle WH\rangle\ v) \subseteq v \text{ and} \\ \qquad\qquad\quad \langle WH\rangle\ (!\ \langle c\rangle) \end{array}$$

Here k is a constant[2], v is a term in the logic, and $\subseteq$ is a BL term denoting a relation. Note that we have assumed a fairly powerful base logic. In particular we must be able to express the first hypothesis formally, and to prove it for some interesting relations $\subset$.

[2]Formally k is a term in BL which thus does not depend on the state.

The rule can be read as: If the initial value of v is less than k, and execution of the loop body strictly decreases v, then the final value of v is no larger than its original value, and after execution the condition c evaluates to false.

We could just as easily have arrived at this rule using the opposite approach — we could have started with a conventional formulation of the rule using both an invariant, and a variant term[3] producing a result in some well founded partially ordered set. We would then have transformed this into the above rule by observing, as we will below, that the part of the rule dealing with invariants is redundant. Thus we will refer to v as the variant term.

At this point the reader may still have trouble believing that the above rule is sufficiently general. Actually it is at least as general as the invariant formulation, provided the base logic is sufficiently expressive. To show this, assume we have a Dijkstra style proof of the while loop. This means we have shown that a preserves some invariant assertion I, and decreases the value of the positive integer-valued[4] variant term f. Thus formally we have

$$\begin{array}{lll} I \wedge \langle c \rangle & \Rightarrow & \langle a \rangle\, I \\ I & \Rightarrow & f > 0 \\ I & \Rightarrow & \langle a \rangle f < f \end{array}$$

This can be transformed into a proof using our rule as follows: Let

[3]Our notion of a variant term is frequently referred to as a variant function. Since it is actually a term (whose *meaning*, like that of all other terms is a function from states and environments to values) we use different terminology.

[4]In Dijkstra's original formulation f is only guaranteed to be positive when the condition, or, for guarded commands, one of the conditions, is true. Given such an f we can easily construct an f' satisfying our stronger constraint. Take f' to be f + 1 whenever c holds, and 1 otherwise. It follows immediately that f' is still decreased by each loop iteration.

$$\begin{array}{lll} v & = & f \text{ if } I \text{ is true} \\ & & \infty \text{ otherwise} \\ k & = & \infty \end{array}$$

$i \subseteq j$ iff $i,j \neq \infty$ and $i < j$, or $i \neq \infty$ and $j = \infty$

Clearly $\subseteq$ is a well-founded partial order. Thus the first hypothesis of our rule is satisfied.

Whenever $v \subset k$ the invariant I holds by the construction of v. So by assumption the invariant will hold after the execution of a as well. Thus in this case $v = f$, both before and after the execution of a, and therefore v is decreased by e. So the second hypothesis holds as well. We conclude that after the while loop c is false, and the final value of v is less than the initial one, and therefore finite. The latter is equivalent to saying that the invariant still holds.

As another example of an application of the above rule consider the following. Assume again that we have a variant term f, and that a and c leave term t invariant. We want to show that:

$$\langle \textbf{while } c \textbf{ do } a \textbf{ od} \rangle\, t \;=\; t$$

In this case we let v be the pair (f, t). We define $(a, b) \subset (c, d)$ iff $a < c$ and $b = d$. Thus values of v with different t components are incomparable.

We can now apply the while-rule in a straightforward manner. (k can be (k', t) for any k' larger than the initial value of f. If necessary we can again introduce a constant ∞.) Our desired result is equivalent to the conclusion

$$\langle \textbf{while } c \textbf{ do } a \textbf{ od} \rangle\, v \;\subseteq\; v$$

of the proof rule.

Side-effects in the condition add some complexity. The easiest way to deal with them is to consider a new programming language construct

while' c **do** a **od** (abbreviated as WH')

It is executed like a **while**-loop, except that after the execution is completed we back up to the state immediately preceding the last execution of c. This is somewhat simpler to analyze than the original **while** construct since its execution is equivalent to an integral number of executions of c; a. Thus it can be axiomatized as before simply by requiring that c; a, rather than just a, decreases the variant v:

[WH' ef rule]:

$$\begin{array}{l} \subset \text{ is a well-founded partial order} \\ \langle c\rangle \wedge v \subset k \Rightarrow (\langle c; a\rangle\ v) \subset v, \\ \vdash \\ v \subset k \Rightarrow (\langle WH'\rangle\ v) \subseteq v \wedge \\ \qquad\qquad \langle WH'\rangle\ (!\ \langle c\rangle) \end{array}$$

We can now define the effect of the real **while**-loop in terms of the **while'** construct by adding the following conclusion to the above rule:

[WH ef rule]:

$$\langle WH\rangle\ E \;=\; \langle WH'; c\rangle\ E$$

Note this is invalid as an independent axiom since the loop must terminate for it to hold.

Soundness Proof: We first have to give a meaning to the WH' construct. As in the definition of the meaning of the normal loop in chapter 2 we let

$$M_{body}[e,s] = M_s[\![a]\!][e, M_s[\![c]\!][e,s]]$$

$$M^0_{body}[e,s] = s$$

$$M^{i+1}_{body}[e,s] = M_{body}[e, M^i_{body}[e,s]]$$

We then define M^*_i as we defined the extended meaning functions M_i in chapter 2, except that we delete the last application of $M_v[\![c]\!]$. That is, if there is a least $j < i$ such that $M_v[\![c]\!][e, M^j_{body}[e,s]$ is **false** then $M^*_i[e,s]$ is $\{M^j_{body}[e,s]\}$, and otherwise $M^*_i[e,s]$ is S', where S' is the set of all conceivable final states, as defined in chapter 2. The rest of the construction proceeds exactly as for the real loop.

We next show that the hypotheses of the rule guarantee that M^*_i is a singleton set, rather than S', assuming i is sufficiently large. Assume this to be false, i.e. assume $M^*_i[e,s_0]$ is S' for all i, and that $v \subseteq k$ holds in environment e and state s_0. Assume further that the two hypotheses of the rule are valid. Let

$$s_j = M^j_{body}[e,s_0]$$

for a given state s_0. Let v_j be the interpretation of v in the state s_j; that is

$$v_j = I_v^{e,s_j}$$

For all states s in which

$$I_v^{e,s} \subseteq^5 k \;\wedge\; M_v[\![c]\!][e,s]$$

the second hypothesis tells us that

$$I_v^{e,M_{body}[e,s]} \subseteq I_v^{e,s}$$

for all states s. We know that s_0 satisfies

[5] Technically we should write $I_{\subseteq}$ and I_k instead of $\subseteq$ and k. This should be apparent from context. The shorthand notation hopefully provides for better readability.

$$\Gamma_v^{e,s_0} \subseteq k \wedge M_v[\![c]\!][e,s_0]$$

We can rewrite the first half as

$$v_0 \subseteq k$$

By assumption we have

$$M_v[\![c]\!][e,s_i]$$

for all i. Using this and the second hypothesis, we obtain by induction

$$v_i \subseteq v_{i-1} \subseteq k$$

Thus the v_i form an infinite descending chain, contradicting the assumption that $\subseteq$ is well-founded.

We may thus assume that whenever the hypotheses hold and s_0 satisfies $v \subseteq k$ then there is a least i such that $M_i^{*}[e,s_0]$ is a singleton set, and thus that $M_s[\![WH']\!][e,s]$ is the single element of that set, which is just the s_i we defined above.

To show that $(<WH'>v) \subseteq v$ is a valid conclusion we have to prove that

$$\Gamma^{e,s_0}_{<WH'>v} \subseteq \Gamma_v^{e,s_0}$$

or, if we use the preceding definition of v_i

$$v_i \subseteq v_0$$

The same inductive proof we used above for this statement still holds, since $M_v[\![c]\!][e, s_j]$ is true for all $j < i$.

The second half of the conclusion just states that $M_v[\![c]\!][e,s_i]$ is false.

We can apply the same reasoning as above to show that under the given hypotheses M_i as defined in chapter 2 must also be a singleton set for sufficiently

large i. The element of that set differs from the element of M_i^{*} only in the final application of $M_s[\![c]\!]$ to the state. It follows that

$$\begin{aligned} M_s[\![WH]\!][e,s] &= M_s[\![c]\!][e, M_s[\![WH']\!][e,s]] \\ &= M_s[\![WH'; c]\!][e,s] \end{aligned}$$

By [< > t meaning thm] this implies the conclusion of [WH ef rule].

As usual, we conclude by giving a value axiom:

[WH val ax]:

$$<\textbf{while}\ c\ \textbf{do}\ a\ \textbf{od}> \ = \ \text{false}$$

Soundness Proof: The soundness of the value axiom is trivial.

4.7. Variable Allocation

We axiomatize the storage allocation functions NewT[]. Although the following axioms are stated for the general version of NewT which requires an argument, they will be applied primarily in the simple case in which the argument is 1. As for assignment, we give separate axioms describing the effect of NewT on various constructs that may follow it. NewT has no effect on values yielded by V. Similarly, it has no effect on allocation functions for another type T'. Thus we may write:

[New ax (1,2)]:

$$\begin{aligned} x \neq <NewT[k]> &\Rightarrow <NewT[k]; V[x]> = <V[x]> \\ x = <NewT[k]> &\Rightarrow <NewT[k]; V[x]> = C_T \end{aligned}$$

$$<NewT[k]; NewT'[j]> = <NewT'[j]>$$

C_T is a type T constant denoting the value c_i used for initializing new locations.

s_a only at T. Since the meaning of NewT'[j] depends only on $s_a[T']$, part (2) is also sound.

If <NewT[]> = Tvar[i] in the current state s, we know from the meaning of NewT and Tvar that

$$A_T[s_a[T] + 1] = A_T[i]$$

Since A_T is 1-to-1 it follows that

$$s_a[T] = i - 1$$

Thus

$$\begin{aligned} I^{e,s}_{<NewT[k];\, NewT[]>} &= M_v[\![NewT[]]\!][e, (s_m, \{T \leftarrow (i\text{-}1)+M_v[\![k]\!][e,s]\} s_a)] \\ &= A_T[(\{T \leftarrow (i + M_v[\![k]\!][e,s] - 1\} s_a)[T] + 1] \\ &= A_T[i + M_v[\![k]\!][e,s]] \end{aligned}$$

The latter is the meaning of Tvar[i + <k>]. The proof of the last line is virtually identical. ●

Since we are actually interested in equalities between variables, and not equalities involving the somewhat contrived function Tvar, we need axioms to relate the two. These just state that Tvar is 1-to-1 and onto.

[Tvar ax]:

$$(i = j) \;=\; (\text{Tvar}[i] = \text{Tvar}[j])$$

$$\forall x\colon \textbf{var}\ T \,.\, \exists i \,.\, \text{Tvar}[i] = x$$

Soundness Proof: Follows directly from the assumption that A_T is 1-to-1 and onto.

●

The above treatment of variables is the only one known to the author to be complete in the sense of chapter 5. In many respects however it is much too specific[7].

It is not too hard to derive some higher level rules which suffice under normal circumstances. We can define new primitives Is_allocated_T[x] which, for a given x with signature **var** T, yields true iff

$$<\text{NewT}[]> = \text{Tvar}[i] \wedge x = \text{Tvar}[j]^8 \;\Rightarrow\; j < i$$

Since the type T is usually apparent from the signature of the argument it will frequently be omitted (as for VT).

We obtain

4.7.3 **Theorem** [Is_allocated thm]:

$$x = <\text{NewT}[]> \;\Rightarrow\; <\text{NewT}[]>\ \text{Is\_allocated}[x]$$

$$\text{Is\_allocated}[x] \;\Rightarrow\; <\text{NewT}[]> \neq x$$

Proof:

Immediate. ●

The former will allow us to conclude that a variable x "is allocated" immediately after a declaration like

let x = NewT[] **in** ...

---

[7] In particular it suffices (and for purposes of the next chapter must suffice) for reasoning about formulas such as

< **while** c **do** a **od**; NewT[]> = < **while** c' **do** a' **od**; NewT[]>

The truth of such a formula in reality depends on the garbage collection algorithm used, in addition to a number of other factors. Furthermore it is not clear why anyone would care whether or not the above formula is true.

[8] By the second half of [Tvar ax] there has to be a j for which the equality holds.

The first axiom is presented only for simple variables x. However it generalizes easily to side effect free expressions:

4.7.1 Theorem [$\langle New \rangle V$ thm]:

Let a be an expression such that $\langle NewT[k]; a \rangle = \langle a \rangle$ and such that $\langle a \rangle V[x] = V[x]$ for any x. Then

$$\langle a \rangle \neq \langle NewT[k] \rangle \Rightarrow \langle NewT[k]; V[a] \rangle = \langle V[a] \rangle$$

$$\langle a \rangle = \langle NewT[k] \rangle \Rightarrow \langle NewT[k]; V[a] \rangle = C_T$$

Proof:

We prove only the first statement. The proof of the second one is similar. By [V comp thm]

$$\langle NewT[k]; V[a] \rangle = \langle V[NewT[k]; a] \rangle$$

By [V arg ax]

$$x = \langle NewT[k]; a \rangle \Rightarrow \langle V[NewT[k]; a] \rangle = \langle NewT[k]; a; V[x] \rangle$$

$$\Rightarrow \langle V[NewT[k]; a] \rangle = \langle NewT[k]; V[x] \rangle$$

By a similar argument we get

$$x = \langle a \rangle \Rightarrow \langle V[a] \rangle = \langle V[x] \rangle$$

Since by assumption $\langle NewT[k]; a \rangle = \langle a \rangle$ we get from the former statement

$$x = \langle a \rangle \Rightarrow \langle V[NewT[k]; a] \rangle = \langle NewT[k]; V[x] \rangle$$

Since we assume $\langle a \rangle \neq \langle NewT[k] \rangle$ we also have $x \neq \langle NewT[k] \rangle$ and thus by [New ax 1]

$$x = \langle a \rangle \Rightarrow \langle V[NewT[k]; a] \rangle = \langle V[x] \rangle$$

We get

$$x = \langle a \rangle \Rightarrow \langle V[a] \rangle = \langle V[NewT[k]; a] \rangle$$

The desired conclusion follows since x does not occur on the right. ●

We can only specify the effect of NewT[k] on another application of NewT in terms of the l-value yielded by the original call. Even in this case we need to introduce some new logical functions.

4.7.2 Definition [Tvar def]:

For each type T there is a function Tvar in the logic. The interpretation of Tvar[i], where i is a natural number, is $A_T[i]$, that is the i^{th} location allocated by NewT[].

We can now specify that the value produced by NewT after NewT[k] is the k^{th} element of the sequence produced by Tvar after the one that would have been produced in the original state. The following axioms express this formally:

[New ax (3)]:

$$\langle NewT[] \rangle = Tvar[i] \Rightarrow \langle NewT[k]; NewT[] \rangle = Tvar[i + \langle k \rangle]$$

$$\langle NewT[] \rangle = Tvar[i] \Rightarrow \langle NewT[k] \rangle = Tvar[i + \langle k \rangle - 1]$$[6]

There is no separate value axiom since the preceding axiom essentially serves that purpose.

Soundness Proof: We prove the soundness of the three parts of [New ax]. First observe that NewT[k] affects only s_a and not s_m. Since the meaning of V[x] does not depend on s_a, the soundness of part (1) is immediate. Furthermore NewT[k] changes

[6] The axioms are stated in a form which makes them convenient to apply in chapter 5. In spite of their similarity, the two lines are independent. The first line holds if we model NewT[k] as always returning the next available location, independent of k, so long as the allocation component of the state gets updated correctly. The second line holds if we model NewT[k] as always returning Tvar[k].

The latter rule tells us that NewT produces a previously inaccessible location, that is one that is not already associated with a previously allocated variable.

We need one additional axiom to allow us to do most reasoning about aliasing and variables without reference to Tvar. We need to know that an allocated variable stays allocated. The simplest way to phrase this is

[Is_allocated ax]:

$$\text{Is\_allocated}[x] \Rightarrow \langle a\rangle\ \text{Is\_allocated}[x]$$

It is clearly sound iff we insist that [s monotonicity thm] hold for all constructs (including the nonterminating ones).[9] It again has the disconcerting property that it is somewhat dependent on detailed (and unobservable) properties of the storage allocator. We will illustrate this less formal development in the next section.

4.8. Simple Declarations

In the following three sections we deal with various mechanisms in our Russell subset for introducing new identifiers. In axiomatizing these constructs it will be essential to allow renaming of identifiers bound inside Russell expressions. We obtain the following three renaming axioms for simple blocks, function constructions, and recursive declarations respectively. We assume that neither y nor any y_i appears in a or $\bar{b}$.

[let rename ax]:

$$\langle \text{let } x == b \text{ in } a \text{ ni}\rangle \equiv \langle \text{let } y == b \text{ in } a\{x \leftarrow y\} \text{ ni}\rangle$$

[9] The soundness of none of the other axioms depended on this property. It follows that this axiom is independent of the others.

[func rename ax]:

$$\langle \text{func } [\bar{x}]\ \{\ a\ \}\rangle \equiv \langle \text{func } [\bar{y}]\ \{\ a\{\bar{x} \leftarrow \bar{y}\}\ \}\rangle$$

[letrec rename ax]:

$$\begin{aligned}&\langle \text{letrec } \bar{x} == \bar{b} \text{ in } a \text{ ni}\rangle \equiv\\ &\langle \text{letrec } \bar{y} == \bar{b}\{\bar{x} \leftarrow \bar{y}\} \text{ in } a\{\bar{x} \leftarrow \bar{y}\} \text{ ni}\rangle\end{aligned}$$

Soundness Proof: We show the soundness of [let rename ax]. Let s' abbreviate M[[b]][e,s]. Using [renaming thm], [e compactness thm], and the definition of let we have

$$\begin{aligned}M[\![\text{let } x == b \text{ in } a \text{ ni}]\!][e,s] &= M[\![a]\!](\{x \leftarrow M_v[\![b]\!][e,s]\}e,\ s')\\ &= M[\![a\{x \leftarrow y\}]\!](\{y \leftarrow e[x]\}\{x \leftarrow M_v[\![b]\!][e,s]\}e,\ s')\\ &= M[\![a\{x \leftarrow y\}]\!](\{x,y \leftarrow M_v[\![b]\!][e,s]\}e,\ s')\\ &= M[\![a\{x \leftarrow y\}]\!](\{x \leftarrow M_v[\![b]\!][e,s]\}e,\ s')\\ &= M[\![\text{let } y == b \text{ in } a\{x \leftarrow y\} \text{ ni}]\!][e,s]\end{aligned}$$

The proofs for [letrec rename ax] and [func rename ax] differ primarily in that [renaming thm] needs to be applied repeatedly, once for each renamed identifier. ●

We finally return to consider the block

let x == b in a ni

which we will abbreviate as LET.

We may assume that the only occurrences of x in the hypotheses of the following inference rules are within the let block, that is within a. We can use the preceding axioms to achieve this.

Now the let rule is easy to express:

[let ax]:

$$x = \langle b\rangle \Rightarrow \langle \text{LET}\rangle \equiv \langle b; a\rangle$$

Soundness Proof: If $e[x] = M_v[\![b]\!][e,s]$ and s' abbreviates $M[\![b]\!][e,s]$ then

$$\begin{aligned} M[\![\text{let } x == b \text{ in } a \text{ ni}]\!][e,s] &= M[\![a]\!][\{x \leftarrow M_v[\![b]\!][e,s]\}e,s'] \\ &= M[\![a]\!][e,s'] \\ &= M[\![b;\, a]\!][e,s] \quad \bullet \end{aligned}$$

In general this rule will be applied as follows: Let $P(\alpha)$ be a formula in which the identifier α occurs only in Russell terms of the form $<\alpha>$ or $<\alpha; d>$. We then use P(b;a) and P(LET) to denote P(y) with b;a and LET, respectively, substituted for α. Assume we wish to prove P(LET). We first prove something of the form

$$x = <b> \quad \Rightarrow \quad P(b;\, a)$$

We assume x does not occur in b or $P(\alpha)$. We then use the preceding axiom to derive

$$x = <b> \quad \Rightarrow \quad P(LET)$$

Using [let rename ax] we can rewrite P(<LET>) so that x no longer occurs in P(<LET>). The above formula can then be simplified to

$$P(LET)$$

Effectively what we have done is translate the binding mechanism of Russell into the predicate calculus binding mechanism, where we presumably already know how to handle it.

Let us consider an example. Assume we want to prove that the following section of code interchanges the values of x and y:


```
let
   t == NewInt[]
in
   t := V[x];
   x := V[y];
   y := V[t]
ni
```

In practice we would outline such a proof by annotating the program with assertions, as is conventionally done with Hoare's logic. In this case however we will be more explicit in the use of our logic. Let us abbreviate the above program segment as α, and the body of the let block as β. We wish to prove

$$\text{Is\_allocated\_Int}[x] \wedge \text{Is\_allocated\_Int}[y] \quad \Rightarrow \quad <\alpha> V[x] = V[y] \wedge <\alpha> V[y] = V[x]$$

Note that this holds whether or not x and y alias. Using the above strategy we only have to prove

$$\begin{aligned} &t = <\text{NewInt}[]> \wedge \text{Is\_allocated}[x] \wedge \text{Is\_allocated}[y] \\ \Rightarrow\ &<\text{NewInt}[];\, \beta> V[x] = V[y] \wedge <\text{NewInt}[];\beta> V[y] = V[x] \end{aligned}$$

By [Is_allocated thm] we obtain

$$\begin{aligned} &t = <\text{NewInt}[]> \wedge \text{Is\_allocated}[x] \wedge \text{Is\_allocated\_Int}[y] \\ \Rightarrow\ &t \neq x \wedge t \neq y \end{aligned}$$

We next simplify $<\beta> V[x]$. We have to distinguish between the two cases $x = y$ and $x \neq y$, that is we have to treat the aliased case separately. We start with the unaliased case.

We let U abbreviate

$$x \neq y \wedge x \neq t \wedge y \neq t$$

We have $<a> U = U$ for any a. Thus

$$U \Rightarrow \langle y := V[t]\rangle V[x] = V[x]$$

$$\langle x := V[y]\rangle\ (U \Rightarrow \langle y := V[t]\rangle V[x] = V[x])$$

$$U \Rightarrow \langle x := V[y];\ y := V[t]\rangle V[x] = \langle x := V[y]\rangle V[x]$$

$$U \Rightarrow \langle x := V[y];\ y := V[t]\rangle V[x] = V[y]$$

$$\langle t := V[x]\rangle\ (U \Rightarrow \langle x := V[y];\ y := V[t]\rangle V[x] = V[y])$$

$$U \Rightarrow \langle \alpha\rangle V[x] = \langle t := V[x]\rangle V[y]$$

$$U \Rightarrow \langle \alpha\rangle V[x] = V[y]$$

Now consider the case in which x and y alias. Let A represent

$$x = y \wedge x \neq t \wedge y \neq t$$

We get

$$A \Rightarrow \langle y := V[t]\rangle V[x] = V[t]$$

$$\langle x := V[y]\rangle\ (U \Rightarrow \langle y := V[t]\rangle V[x] = V[t])$$

$$A \Rightarrow \langle x := V[y];\ y := V[t]\rangle V[x] = \langle x := V[y]\rangle V[t]$$

$$A \Rightarrow \langle x := V[y];\ y := V[t]\rangle V[x] = V[t]$$

$$\langle t := V[x]\rangle\ (U \Rightarrow \langle x := V[y];\ y := V[t]\rangle V[x] = V[t])$$

$$A \Rightarrow \langle \alpha\rangle V[x] = \langle t := V[x]\rangle V[t]$$

$$A \Rightarrow \langle \alpha\rangle V[x] = V[x]$$

$$A \Rightarrow \langle \alpha\rangle V[x] = V[y]$$

The last step follows from x = y and [V = thm]. Using [New ax] we obtain

$$\langle NewInt[];\ V[y]\rangle = V[y]$$

Combining the preceding three results we get

$$A \vee U \Rightarrow \langle NewInt[];\ \alpha\rangle V[x] = V[y]$$

Using an almost identical proof we have

$$A \vee U \Rightarrow \langle NewInt[];\ \alpha\rangle V[y] = V[x]$$

We previously showed that

$$t = \langle NewInt[]\rangle \wedge Is\_allocated\_Int[x] \wedge Is\_allocated\_Int[y]$$
$$\Rightarrow A \vee U$$

This concludes the proof.

4.9. Function Constructions and Calls

We start with a simple effect axiom for function constructions:

[func ef ax]:

$$\langle func[\hat{x}]\ \{a\};\ b\rangle = \langle b\rangle$$

Soundness Proof: Follows immediately from the meaning definition for function constructions. ●

We can give a detailed description of the value returned by a function construction only by describing its behaviour in a function call. We can however give a very general value axiom which tells us that the value produced by a function construction, and indeed any expression with function signature, is independent of the state:

[fn val ax]:

$$\langle a;\ b\rangle = \langle b\rangle$$

whenever b has function signature.

Soundness Proof: By [s compactness thm], the value component of the meaning of an expression with func signature is independent of the state. ●

We present the function call axiom in four parts. The first rule is essentially η-reduction in the lambda calculus. It deals with the case in which the environment is unaffected by argument parameter binding. Usually this will be used to prove that a given function satisfies certain properties, independent of any particular application. The remaining three axioms then allow us to make use of these general properties in proving statements about a specific application.

[η axiom]:

$$<(\mathbf{func}\boxed{\bar{x}}\ \{a\})\ \boxed{\bar{x}}> \;\equiv\; <a>$$

Soundness Proof: Let f be $M_v[\![\mathbf{func}\boxed{\bar{x}}\ \{a\}]\!][e,s]$. First observe that

$$M_v[\![x_1; ...; x_i]\!][e,s] = e[x_i]$$

for any i. Thus the value of the i^{th} argument is just $e[x_i]$. This gives us

$$\begin{aligned} M[\![(\mathbf{func}\boxed{\bar{x}}\ \{a\})\ \boxed{\bar{x}}]\!][e,s] &= f[M_s[\![\boxed{\bar{x}}]\!][e,s], e[x_1], ..., e[x_n]] \\ &= f[s, e[x_1], ..., e[x_n]] \\ &= M_v[\![\mathbf{func}\boxed{\bar{x}}\ \{a\}]\!][e,s][s, e[x_1], ..., e[x_n]] \\ &= (\lambda s \lambda \bar{y} \,.\, M[\![a]\!][\{\bar{x} \leftarrow \bar{y}\}e, s])[s, e[x_1], ..., e[x_n]] \\ &= M[\![a]\!][\{\bar{x} \leftarrow (e[x_1], ..., e[x_n])\}e, s] \\ &= M[\![a]\!][e, s] \quad \bullet \end{aligned}$$

In general function applications will of course not explicitly give the function construction. Instead functions are more likely to be bound to identifiers. We can deduce the equality between these identifiers and the function construction from the applicable declaration axiom. What we need then is an axiom allowing us to deduce the equality of applications from the equality of the operators.

[fn subst ax]:

$$<a> = <b> \;\Rightarrow\; <a\boxed{\bar{d}}> \equiv <b\boxed{\bar{d}}>.$$

Soundness Proof: The meaning of the applications depends only on the value component of the meaning function for the operator and on the meaning of the operands. Since the operands of the two applications are identical and $<a> = <b>$ guarantees that the value components of the operator meanings are the same, we can conclude that the meaning of the two applications is the same. $\bullet$

The following group of axioms tell us that the meaning of an application is dependent only on the allocation component of the state[10] and the values of the variables passed as parameters.

Consider the application $f\boxed{\bar{x}}$ where f is an expression with function signature, and $\bar{x}$ is a list of identifiers. Let $\bar{z}$ be the sublist of the list of argument identifiers $\bar{x}$ which have **var** signature.

[appl state ax]:

$$\begin{aligned} &\wedge_j (<a; NewT_j[]> = <b; NewT_j[]>) \wedge \wedge_i (<a; V[z_i]> = <b; V[z_i]>) \\ &\Rightarrow\; <a; f\boxed{\bar{x}}> = <b; f\boxed{\bar{x}}> \wedge <a; f\boxed{\bar{x}}; NewT_j[]> = <b; f\boxed{\bar{x}}; NewT_j[]> \end{aligned}$$

$$\begin{aligned} &\wedge_j (<a; NewT_j[]> = <b; NewT_j[]>) \wedge \wedge_i (<a; V[z_i]> = <b; V[z_i]>) \\ &\qquad \wedge (<a; V[x]> = <b; V[x]>) \\ &\Rightarrow\; <a; f\boxed{\bar{x}}; V[x]> = <b; f\boxed{\bar{x}}; V[x]> \end{aligned}$$

---

[10] The dependence on the allocation component introduces a lot of the apparent complexity into the following [appl state ax]. Again this is due largely to our desire to obtain a system for which we can prove some kind of completeness result. The dependency is unlikely to exist for real function definitions, with the following two exceptions. The value returned by NewT following an application will depend on the allocation component of the state prior to the application. If the function in question returns a newly allocated location, that location will depend on the allocation component. If we use the "axiomatization" of NewT using Is_allocated these dependencies should be insignificant, in that we are not concerned with the exact location produced in any case. We can guarantee that an r-value or function produced by an application, or the value of an argument variable does not depend on the allocation component of the state.

Also note that only the allocation components of the argument types and result type are relevant. Thus the hypotheses in the following axiom could be weakened. For our present formal treatment this only complicates matters. Since this is not a rule we would want to apply directly in practice, we are not concerned with anything else. (It is essential that the rule be weakened if we consider a language with an infinite collection of types, as in parts of chapter 6.)

Soundness Proof: Soundness follows directly from [s compactness thm] applied to the set of identifiers $\{z_i\}$ or in the latter case, $\{z_i\} \cup \{x\}$. Note that we insure that allocation components of the state after executing a and b respectively are always equal; therefore we do not need to guarantee that identifiers in these sets are bound to allocated locations. By restricting the arguments to identifiers we insure that they alias in the same way in both cases. ●

To illustrate one use of this axiom we introduce

4.9.1 **Definition** [gen appl def]:

By the *general application* of f to $\bar{x}$ with respect to the lists of identifiers $\bar{w}$ and $\bar{y}$, written

$$[\bar{w},\bar{y}]\langle f[\bar{x}]\rangle$$

we mean

$$\langle z_1 := w_1; \ldots; z_m := w_m; \mathrm{NewT}_1[y_1]; \ldots; \mathrm{NewT}_k[y_k]; f[\bar{x}]\rangle$$

or the application of f after y_i new locations of the i^{th} type T_i have been allocated, and after the z_i have been assigned values w_i.

We can thus state that some property G holds of the value produced by the application of f in any state by saying that it holds for the general application and all $\bar{w}$ and $\bar{y}$. We can use a similar approach to describe the state after the application. This allows us to express (and hopefully prove) a general property G(f) of the function f. The preceding axiom allows us to infer

$$G(f) \Rightarrow \langle a\rangle G'(f)$$

and thus apply this general property to a specific state. This is not otherwise possible since we have no direct way of stating inside G that a property holds "in all states", but only that it holds for all values of the variables appearing in the application.

We finally describe parameter passing in a way completely analogous to the axiom for the **let** construct.

[$\simeq$ def]:

Let $\bar{x}$ be a list of identifiers and let $\bar{b}$ be a list of Russell expressions. Assume they both have length n. We then define

$$\bar{x} \simeq \bar{b}$$

to mean

$$x_1 = \langle b_1\rangle$$

$$x_2 = \langle b_1; b_2\rangle$$

$$\vdots$$

$$x_n = \langle b_1; b_2; \ldots; b_n\rangle$$

Using this notation, and again making the assumption that no x_i appears in any b_i, the argument-parameter binding axiom is:

[arg ax]:

$$\bar{x} \simeq \bar{b} \Rightarrow \langle f[\bar{b}]\rangle \equiv \langle b_1; \ldots; b_n; f[\bar{x}]\rangle$$

Soundness Proof: Fix an environment e and state s in which $\bar{x} \simeq \bar{b}$ holds. Then let

$$s_0 = s$$

$$s_{i+1} = M_s[\![b_{i+1}]\!][e, s_i]$$

$$v_{i+1} = M_v[\![b_{i+1}]\!][e, s_i]$$

as we did when we defined the meaning of the function call. Thus $\bar{x} \simeq \bar{b}$ implies

$$
\begin{aligned}
I^{e,s}_{x_i} &= e[x_i] \\
&= I^{e,s}_{<b_1; \dots ; b_i>} \\
&= I^{e,s_{i-1}}_{<b_i>} \\
&= v_i
\end{aligned}
$$

We then have

$$
\begin{aligned}
M[\![f[\bar{b}]]\!][e,s] &= (M_v[\![f]\!][e,s])[s_n, \bar{v}] \\
&= (M_v[\![f]\!][e,s])[s_n, e[x_1], \dots, e[x_n]] \\
&= (M_v[\![f]\!][e,s_n])[s_n, e[x_1], \dots, e[x_n]] \text{ ([s compactness thm])} \\
&= M[\![f[\bar{x}]]\!][e,s_n] \\
&= M[\![b_1; \dots; b_n; f[\bar{x}]]\!][e,s] \quad \bullet
\end{aligned}
$$

The reader may wish to compare this with a more conventional treatment. See for example [Gri 80] or [Rey 81].

4.10. Recursive declarations

We consider the block

$$\textbf{letrec } \bar{x} == \bar{b} \textbf{ in } a \textbf{ ni}$$

which we will abbreviate as LETREC.

As for the loop our proof rule will be stated so that we can't prove anything nontrivial about the construct unless it terminates.[11] Unlike the loop, it will be necessary to talk about termination explicitly. Thus we would like to introduce a predicate

$$T(\alpha)$$

which is true in exactly those states in which the Russell expression α terminates[12].

It is easy enough to introduce T as a new primitive. We would need meaning functions to tell us not only the state and value produced by evaluation of an expression, but also whether or not it terminated[13]. We then interpret $T(\alpha)$ to be true iff the "termination component" of $M[\![\alpha]\!][e,s]$ is true. We would further need a collection of rules to reason about $T(\alpha)$. For most constructs this rule would just state that the construct terminates whenever its components do. For the loop it would state that the loop terminates if the subexpressions do and if the hypotheses of [WH' ef rule] are satisfied.

This preceding approach is clearly the one to be pursued for "practical" reasoning about programs. Unfortunately it has two disadvantages: First, it is tedious to develop the additional formalism. Second, we are forced to introduce another primitive in the logic. Thus we avoid this development by noting that something akin to termination is already expressible in the logic we are developing. That is, it is possible to write down a formula which is globally valid iff some construct α terminates.[14]

We give a formula $T(\alpha)$ such that $T(\alpha)$ is guaranteed to hold in states in which α terminates. Furthermore if α does not terminate, then there is an allowable mean-

[11] Recall that for the loop construct we do not insist that the loop as a whole terminate. For example [WH ef rule] and [WH' ef rule] suffice to prove

$$< \textbf{while } V[x] \textbf{ do } \beta; x := \textbf{false od} > (V[x] = \textbf{false})$$

whether or not β terminates. Our letrec semantics require that subexpressions terminate. This facilitates the following development.

If we did not require subexpressions to terminate we would have to explicitly introduce the concept of "termination relative to termination of subexpressions". This appears unnecessarily complicated.

[12] We have not formally defined what it means for an expression α to terminate in environment e and state s. We take this to mean that all extended meanings used in inductively building up $M[\![\alpha]\!][e,s]$ produced singleton states. It should be clear how to turn this into a formal inductive definition.

[13] This would be one way of formalizing the notion of termination anyway.

[14] Note however that nontermination is not expressible. In particular if the construct α does not terminate we will neither be able to prove nor disprove the formula we construct.

ing for α such that $T(\alpha)$ is false. For this purpose it will be useful to assume that we have a type E which is only mentioned by the formulas $T(\alpha)$. In particular it never appears in the signatures of Russell expressions (other than those introduced below to state $T(\alpha)$). We assume that R_E contains at least two values t and n. Informally we will use the fact that n was stored at a location of type E to indicate that a construct failed to terminate. We may simply introduce a new type (with no operations on it) for this purpose.

A proof almost identical to that of [s compactness thm] tells us that if execution of α terminates in the current state, then it cannot change the values stored at the locations L_E. Thus if x has signature **var** E we have

$$<w := t; \alpha; V[w]> = t$$

On the other hand, if α fails to terminate in a state s, then we are free to chose a meaning function for it that always stores n in some (or all) of the locations L_E[15].

We define the termination predicate $T(\alpha)$ as follows:

$$T(\alpha) \quad = \quad \forall w : \mathbf{var}\ E\ .\ <w := t; \alpha; V[w]> = t$$

As we argued above, $T(\alpha)$ will hold for all allowable assignments of meaning functions and for all states only if α terminates. In particular we will be able to prove $P \Rightarrow T(\alpha)$ only if α terminates in all states satisfying P[16].

Since we wish to think of T as indicating termination, we need to modify the semantics defined in chapter 2 slightly. This true if all nonterminating constructs as

[15]We have to do this consistently for all such states and environments, so that we don't violate the theorems in chapter 2. This causes no difficulties.

[16]This is technically true only if P does not nention E and therefore does not restrict the set of allowable meanings.

changing the value at some type E location to n. As it stands this is an allowable interpretation. From now on, we will restrict our notion of an allowable interpretation so that nonterminating constructs *must*[17] behave this way. Recall that this kind of condition does not constrain a language implementation, since it affects only nonterminating constructs. We are simply constructing a convenient model to show that our theory is consistent and gives reasonable results in those cases in which it matters. We make no claims that this aspect of the model reflects reality.

To prove anything interesting about the **letrec** construct we will first have to prove that in the states in which we are interested the body terminates.

Rather than being allowed to conclude that <LETREC> = <a> under the appropriate conditions, we use the weaker conclusion that LETREC ~ a where a ~ b means roughly that a and b are equivalent *whenever* b *terminates*. Formally we use the following definition:

4.10.1 Definition [~ def]:

We use the notation a ~ b only if a and b have the same signature.

1. If a and b have **val** or **var** signature then a ~ b is defined to be

$$T(b) \Rightarrow a \equiv b$$

2. If a and b have **func** signature then a b means

$$T(b) \Rightarrow (<a>t = <b>t \ \wedge\ a[\bar{x}] \sim b[\bar{x}])$$

[17] Formally we modify our model to reflect this constraint by further restricting the otherwise arbitrary choices for nonterminating constructs. This is analogous to the constraints we imposed to insure the validity of [s monotonicity thm] for nonterminating constructs.

This modification is needed for the following rule to be sound. On the other hand we make no attempt to completely describe this model. In particular we will never be able to prove nontermination of a construct. The theorem in the next chapter is stated to reflect this fact. This approach was chosen since it is relatively simple, and there is rarely a reason to prove that a construct fails to terminate normally.

Clearly we have no hope of proving such a statement, or anything else about a letrec block without being allowed to say something about the x_i. The approach we used for non-recursive declarations clearly fails in the face of the declaration

$$\text{letrec } f == f \text{ in } \ldots$$

Thus the assumption we are allowed to make have to be phrased as something like $x_i \sim b_i$ rather than $x_i = b_i$. This is not quite sufficient, so we define a slightly stronger property:

4.10.2 Definition [~ def]:

We write $\bar{x} \sim \bar{b}$ ($\bar{x}$ is defined to be $\bar{b}$) if each $e[x_i]$ is an acceptable meaning of

$$\text{letrec } \bar{x} == \bar{b} \text{ in } x_i \text{ ni}$$

This is a concise way of stating that x_i is bound to a meaning consistent with the extended meaning for it as defined in the meaning definition for letrec.

We claimed that this was stronger than the ~ relation. we now formalize this claim:

[~ ⇒ ~ ax]:

$$(\bar{x} \sim \bar{b}) \Rightarrow (\wedge_i\, x_i \sim b_i)$$

Soundness Proof: In the following discussion we will not be careful about the distinction between extended domains and their simple counterparts. We will frequently write write down a simple value when we mean the fully defined extended value corresponding to it.

Fix an environment e and a state s. Define $\bar{y}$ as in the definition of the meaning of the letrec construct, that is as the limit of the sequence $\bar{y}^i$ of extended meanings

of the b_j, where each element in the sequence is used as the bindings for $\bar{x}$ to obtain the next element in the sequence.

We argue first that $\bar{y}$ is "a fixpoint of $\bar{b}$". That is,

$$\bar{y}_j = E_v[\![b_j]\!][\{\bar{x} \leftarrow \bar{y}\}e, s]$$ [18]

From the monotonicity of the extended meanings we know that for each i,

$$E_v[\![b_j]\!][\{\bar{x} \leftarrow \bar{y}\}e, s] \geq E_v[\![b_j]\!][\{\bar{x} \leftarrow \bar{y}^i\}e, s] = y_j^{i+1}$$

It is easily shown that ≥ is preserved when we take limits of monotone sequences. Thus we have

$$E_v[\![b_j]\!][\{\bar{y} \leftarrow \bar{x}\}e, s] \geq y_j$$

For the converse we need to show by structural induction on b_j that for a monotone sequence e^i,

$$\mathit{lim}\ E_v[\![b_j]\!][e^i, s] \geq E_v[\![b_j]\!][\mathit{lim}\ e^i, s]$$

where *lim* is the notion of limit introduced in chapter 2. Given this result, we know that

$$y_j = \mathit{lim}\ y_j^{i+1} = \mathit{lim}\ E_v[\![b_j]\!][\{\bar{x} \leftarrow \bar{y}^i e^i, s] \geq E_v[\![b_j]\!][\{\bar{x} \leftarrow \bar{y}\}e, s]$$

The basic argument used in the inductive proof is that whenever the right side (the meaning in the limit environment) or its application is fully defined, then it must already be fully defined in some environment e^i. Thus the left side must also be defined. We omit the details and refer the reader to similar proofs in the literature (cf. [deB 80]).

---

[18] Translated to conventional terminology, we have to prove that extended meanings are continuous.

Thus we know that $\bar{y}$ is a fixpoint of $\bar{b}$ in the preceding sense. Assume that $\bar{x} \sim \bar{b}$ holds in e. From the monotonicity of the meaning functions it follows that for any Russell expression d,

$$M[\![d]\!][e,s] \geq E[\![d]\!][\{\bar{x} \leftarrow \bar{y}\}e,s]$$

Another simple structural induction shows that if d has **val** or **var** signature[19] then the right side of this relation is fully defined whenever the left side terminates (in the sense of not changing an E component of the state).

The soundness of the preceding axiom follows immediately if we use the preceding observation with various applications of the b_i. The soundness of the following axiom also follows immediately. ●

We finally state the rule for the letrec block analogously to the one for a let block:

[letrec ax]:

$$\bar{x} \sim \bar{b} \Rightarrow \text{LETREC} \sim a$$

Most commonly we prove a property of the LETREC construct much as we would prove such a property of the LET construct. Assume we wish to prove

$$Q \Rightarrow P(\text{LETREC})$$

where Q expresses any conditions necessary for termination of the LETREC construct and the LETREC block has **val** or **var** signature.

We first use [letrec rename ax] to insure that there are no occurrences of the x_i in Q, or in P itself, that is outside the LETREC construct. We then prove

[19] This condition is necessary to exclude
letrec f == f in f ni
and similar constructs which are treated as terminating.

$$(Q \wedge \wedge_i x_i \sim b_i) \Rightarrow (P(a) \wedge T(a))$$

using the above axioms to draw useful conclusions from the ~ relations. Usually this will require an inductive proof to show termination of the b_i. Using [~ ⇒ ~ ax] it follows that

$$(Q \wedge \bar{x} \sim \bar{b}) \Rightarrow (P(a) \wedge T(a))$$

From this we get, with the aid of [letrec ax] and [~ def]:

$$(Q \wedge \bar{x} \sim \bar{b}) \Rightarrow P(\text{LETREC})$$

We can then use [letrec rename ax] to remove any occurrences of the x_i from within LETREC. We need the following axiom to insure that there are x_i which have the appropriate relationship to the b_i:

[∃ ~ ax]:

$$\exists \bar{x} . \bar{x} \sim \bar{b} \quad (\text{for any } \bar{b})$$

Soundness Proof: Immediate from the definition of ~. ●

Since there are no longer any occurrences of the x_i in P(LETREC) this allows to remove the conditions $\bar{x} \sim \bar{b}$. Thus we obtain

$$Q \Rightarrow P(\text{LETREC})$$

It is worth noting that our axiomatization of letrec itself is not inelegant. The complexity rests almost exclusively in the model used to prove the axiomatization sound.

CHAPTER 5

A RELATIVE COMPLETENESS PROOF

What follows is a proof that the preceding axiomatization of our Russell subset is relatively complete, roughly in the spirit of [Coo 76] or [Har 79]. Intuitively this can be interpreted to mean that we haven't forgotten any rules in the last chapter, that is there are no "true" statements about Russell (subset) programs that we can't prove because the appropriate rule is missing from chapter 4.

We continue to refer to the base logic defined in chapter 3 as BL. Of the axioms presented in chapter 4 we view only [Tvar ax] as being part of the axiomatization of BL.

We refer to the full logic as RPL (Russell Programming Logic). We assume without loss of generality that we are only dealing with formulas in pure RPL, that is formulas in which constructs of the form <a> t have been eliminated using [<> t def].

It will be useful to define two intermediate logics, as well as a restricted version of BL.

5.0.1 Definition [BL^+ def]:

BL^+ is RPL restricted to formulas in which all Russell terms have the form

<V[x]>

or technically

<TV[x]>

where x is an identifier and T is some type. We use the axiomatization of BL

together with [V = thm] as a BL^+ axiomatization.

5.0.2 Definition [BL^{++} def]:

BL^{++} is RPL restricted to formulas in which all Russell terms have the form

<V[x]>

or

<NewT[]>

where x is an identifier and T is some type. The axiomatization of BL^+ (together with standard logical rules) will suffice as a BL^{++} axiomatization. Note that in BL^{++} we can refer to all of the state, whereas in BL^+ we have no way of expressing anything which depends on the allocation component of the state. In BL we have no way of expressing any dependency on the state of the computation.

5.0.3 Definition [BL^- def]:

A BL^- formula is a BL formula with no occurrences of Tvar.

To state the main theorem we need a pair of definitions:

5.0.4 Definition [compl descr def]:

Let P(x) be a BL^{++} formula in which x is free, that is, does not occur with any quantifier, and does not occur inside Russell terms. We use P(<a>) to denote P(x) with all occurrences of x replaced by <a>. Assume <a> has val or var signature. The pair (x, P(x)) is a *complete description* of the Russell term <a> if

1. P(<a>) holds in those states and environments in which a terminates (i.e. in those states and environments in which for given A_T and L_T the mean-

ing of a is uniquely defined), and conversely,

2. if P(x) is true in state and environment s and e, then a terminates and $x = \langle a \rangle$ holds in s and e.

Informally P(x) states that "$\langle a \rangle$ terminates and produces the value x". The statement that $\langle a \rangle$ has a complete description essentially means that $\langle a \rangle$ can be expressed in BL^{++} (rather than full RPL).

We write the complete description of $\langle a \rangle$ as $(C_{\langle a \rangle}(x), x)$.

We extend this to a term $\langle a \rangle$ with function signature as follows. Let $\bar{x}$ be a sequence of new identifiers with appropriate signatures for the arguments to a. Let f be an identifier with the same signature as $\langle a \rangle$. Let α be the general application (cf. section 4.9)

$$[\bar{w},\bar{y}]\langle a[\bar{x}]\rangle$$

with respect to $\bar{w}$ and $\bar{y}$. Let $\bar{z}$ be the x_i with var signature. Correspondingly let β be the general application of f. (f, P(f)) is a *complete description* of $\langle a \rangle$ if P(f) has the form

$$\forall y_1 \dots \forall y_k \forall x_1 \dots \forall x_n \forall w_1 \dots \forall w_m \,.\, C_{\langle \alpha \rangle}(\langle \beta \rangle) \wedge \bigwedge_j C_{\langle \alpha;\, V[z_j]\rangle}(\langle \beta;\, V[z_j]\rangle) \wedge \bigwedge_T C_{\langle \alpha;\, NewT[]\rangle}(\langle \beta;\, NewT[]\rangle)$$

This is an inductive definition since α may still have function signature and thus $C_{\langle \alpha \rangle}(x)$ may again have the above form[1]. (In this case $C_{\langle \alpha \rangle}(\langle \beta \rangle)$ is not formally syntactically correct since x in $C_{\langle \alpha \rangle}(x)$ will appear in a Russell term. We solve this problem by substituting β instead of $\langle \beta \rangle$ where appropriate.) We

[1] Of course it is necessary to use different names for the bound variables introduced in $C_{\langle \alpha \rangle}(x)$.

refer to the collection of Russell terms appearing in the complete description of a function f as the *extended general applications* of f. Thus the complete description of a Russell term with function signature is just the conjunction of the complete descriptions of its extended general applications. Informally we completely describe a function by completely describing all possible (possibly repeated) applications.

5.0.5 Definition [partial descr def]:

Let P(x) again be a BL^{++} formula in which x is free. Let $\langle a \rangle$ have **val** or **var** signature. (x, P(x)) is a *partial description* iff P(x) holds exactly when either $\langle a \rangle$ does not terminate, or $x = \langle a \rangle$. We generalize this definition to Russell terms of arbitrary signatures as for complete descriptions.

We denote the partial description of $\langle a \rangle$ by $(x, P_{\langle a \rangle}(x))$.

If we consider an $\langle a \rangle$ with **val** or **var** signature then the complete description of $\langle a \rangle$ differs from the partial description only in the case in which $\langle a \rangle$ fails to terminate. Furthermore one is easily expressible in terms of the other. For example

$$P_{\langle a \rangle}(x) = ((\exists y \,.\, C_{\langle a \rangle}(y)) \Rightarrow C_{\langle a \rangle}(x))$$

For what follows, we need a long, though quite natural, list of assumptions:

1. All primitive operations and constants in the programming language which were not explicitly mentioned in the preceding chapter are axiomatized as in section 4.1 and 4.2. (Many generalizations are obviously possible.)

   The names used for those functions in the logic that correspond to built-in side effect free operations in Russell (e.g. '+') are distinct from identifiers introduced elsewhere. All such builtin operations have **val** or **var** parameter and result signatures.

NewT is axiomatized using Tvar. The meaning definitions in chapter 2 are adjusted to allow the introduction of the termination predicate as in the last section of the preceding chapter.

2. We assume the base logic BL is based on ZF set theory with a reasonable definition of quantification over signatures.

Of course much weaker systems would suffice. But the purpose of this chapter is to show in as painless a way as possible that nothing has been omitted from the preceding chapter, and not to characterize the smallest base logic that's suitable for extension to RPL. The particular properties we need are:

a) As in chapter 3 we assume that we can reason about equality of l-values and Russell functions.

b) BL^- includes a conventional axiomatization of the predicate calculus with equality.

b) BL^{++} is expressive for our programming logic RPL (Russell Programming Logic). That is, for every Russell term $<a>$ with val or var signature, there exists a complete description $(x, C_{<a>}(x))$ of $<a>$[2].

[2]This is certainly true if we take something like ZF set theory as BL^-. In BL^{++} we can express a component of the environment by just writing down the name of the identifier, we can refer to a component of the map portion of the state as $<V[x]>$, and finally we can state that i is the T component of the allocation portion of the state as $<NewT[]> = Tvar[i+1]$. We can formally express both the meaning of the given construct, and the fact that we assigned a unique meaning in terms of these values. (We simply formalize the chapter 2 constructions.) $C_a(x)$ then states that x is this meaning. Since we are only dealing with partial recursive functions, we can also argue easily, as in [Coo 76], that theories much weaker than ZF set theory suffice. Note however that we need BL^{++} rather than BL or BL^+ to be able to talk about the current state.

A simple inductive proof on the structure of the signature allows us to generalize this to Russell terms of arbitrary signature. Since partial descriptions are expressible in terms of complete descriptions for val and var terms we can show in the same way that partial descriptions of arbitrary Russell terms also exist.

It follows that there is a BL^{++} formula which is equivalent to a given RPL formula P in those states and environments in which all Russell terms in P terminate[3]. If we restrict ourselves to those states then for any $<a>$ appearing in P, then both $P_{<a>}(x)$ and $C_{<a>}(x)$ are equivalent to $x = <a>$. Thus we can remove undesirable Russell terms $<a>$ from P by substituting a new identifier $x_{<a>}$ for an occurrence of $<a>$, thus obtaining P', and then replacing P by

$$\forall x_{<a>} . C_{<a>}(x_{<a>}) \Rightarrow P'$$

In particular note that if a is a terminating Boolean Russell expression, then $<a>$ is equivalent to

$$\forall x . C_{<a>}(x) \Rightarrow x$$

d) N-tuples are definable if their components are.

e) A relation is definable whenever we can write down the corresponding binary predicate and its domain is definable.[4] As assumed in chapter 3, we can thus reason about mathematical functions in BL.

[3]The generalization to nonterminating constructs is tricky, since the truth of such a formula is dependent on the particular meaning chosen, and the meanings chosen for different nonterminating constructs are not independent. In particular $<a> = <a>$ always holds. Fortunately we won't need such a generalization.

Note that the construction we give is always vacuously true in states in which any of the constructs fail to terminate.

[4]This is much stronger than necessary. We will only construct relations describing the input and output states of computations.

We assume that the signature calculus is extended so that we can assign signatures to arbitrary tuples and relations.

f) The reflexive-transitive closure R^* of a relation R is definable.

g) Well-foundedness of a relation is expressible.

h) The fact that the set of values satisfying a unary predicate is finite is expressible.

i) Induction on well-founded partial orders can be carried out whenever the induction step can be carried out.

j) For the sake of convenience, we explicitly assume that relations expressing calling chains of mutually recursive functions are definable. The last subsection of this chapter will make it clear exactly what this means. (With some care it can be argued that this assumption is redundant.)

3. For purposes of simplicity we assume that any formula in RPL can be written in prenex normal form. (This would only be violated if BL were a constructive logic.) Also for the sake of simplicity we assume that the only operators in the logic which require Boolean arguments are the standard logical connectives and that we can write quantifier free formulas in disjunctive normal form.

4. We assume that the axiomatization of RPL is extended with a complete axiomatization of BL^-, e. g. by adding all true statements of BL^- as axioms.[5] The resulting axiomatization is unlikely to be recursively enumerable, but as in [Coo 76] that will not concern us here.

[5] This need not include those which use (implicit or explicit) quantification over Russell functions.

Of course we have no hope of proving RPL complete without assumption (4), since under reasonable assumptions BL^- will never be complete. Thus introducing this assumption essentially reduces our statement to the weaker claim that the incompleteness of RPL can, in some some sense, be traced back exclusively to the incompleteness of BL^-. Thus we haven't forgotten to specify some obscure part of the language semantics.

We require another definition:

5.0.6 **Definition** [ins term def]:

Fix an assignment of meaning functions to Tvar (A_T) and nonterminating constructs. Given a valid (w.r.t. given meaning assignment) formula P we first write it in prenex normal form, with the non-quantifier part of the formula in disjunctive normal form. We then say that P *insures termination* if it remains valid if we interpret disjuncts as false in any state and environment in which one of the Russell terms fails to terminate. In this definition we treat a while' loop as not terminating if the corresponding while loop does not terminate[6].

In the preceding definition we insisted only that P be valid rather than globally valid. This doesn't matter:

5.0.7 **Theorem** [term $\Rightarrow$ glbl thm]:

Any valid (w.r.t. given assignment of meaning functions) formula that insures termination is globally valid.

Proof:

From the definition of global validity it does not matter what meaning we assign

[6] This avoids the otherwise messy situation in which the last evaluation of the conditional fails to terminate, but the loop does.

to a nonterminating construct. The only other place we allow a non-unique meaning function is for Tvar or more specifically L_T and A_T. Consider two different assignments which (for each T) we denote by (L_T, A_T) and (L'_T, A'_T). For each T we then define a mapping I_T from L_T to L'_T by

$$I_T(A_T(n)) = A'_T(n)$$

Since the only operations involving l-values are = and A_T, both of which are preserved by I_T, the above construction extends in the natural way to an isomorphism between the two models of RPL. Thus a formula is valid in one such model iff it is valid in all such models. ●

5.0.8 Definition [basic statement def]:

An RPL formula is a *basic statement* if all Russell terms appearing in it have **val** or **var** signature and the formula does not involve quantification (explicit or implicit) over function signatures. Note that we do not exclude the definition of functions within Russell terms.

The remainder of this chapter will be devoted to proving

5.0.9 Theorem [RPL rel compl thm]:

Under the above assumptions any valid basic statement which insures termination is provable using the above axiomatization. In particular any Dijkstra-style total correctness assertion

$$P \Rightarrow (\langle a \rangle Q \wedge T(a))$$

is provable.

5.1. Some Lemmas

The following four lemmas take care of some preliminaries. We will then be able to prove our theorem with a single, though rather lengthy, induction.

5.1.1 Theorem [BL compl lemma]:

Under the above assumptions 2 through 4 any globally valid BL formula is provable using the indicated axiomatization (that is the BL^- axiomatization together with [Tvar ax]).

Proof:

For any BL formula P, we derive an equivalent BL^- formula P' s.t. $P' \Rightarrow P$ is provable in BL. If P is valid then P' is provable by assumption 4. Thus P is provable.

To construct P', we first introduce identifiers p_Tvar which intuitively represent the possible functions from nonnegative integers to L_T that each function Tvar may represent. We then write P' as

$$\forall p\_T_1var\text{: }\mathbf{func}[\mathbf{val}\ Int]\ \mathbf{var}\ T\ \ldots$$
$$\forall p\_T_nvar\text{: }\mathbf{func}[\mathbf{val}\ Int]\ \mathbf{var}\ T\ .\ (C_{T_1} \wedge \ldots \wedge C_{T_n}) \Rightarrow Q$$

where Q is P with each occurrence of Tvar replaced by p_Tvar. C_{T_i} expresses the constraints on p_$T_i$var. We insist simply that it be one to one and onto (when treated as a function from the nonnegative integers). Thus C_{T_i} can be formally expressed as

$$\forall j,k\text{: }\mathbf{val}\ Int\ .\ (j \geq 0 \wedge k \geq 0) \Rightarrow (p\_T_ivar[j] = p\_T_ivar) = (i = j)$$
$$\wedge \forall l\text{: }\mathbf{var}\ T_i\ .\ \exists j\text{: }\mathbf{val}\ Int\ .\ j \geq 0 \wedge p\_T_ivar[j] = l$$

Since the acceptable meanings for Tvar (the acceptable A_T) are precisely those allowed for p_Tvar by the formula C_T it follows that P' is valid iff P' is globally valid. P' $\Rightarrow$ P is provable by simply substituting Tvar for each p_Tvar. Each C_{T_i} then reduces to true by [Tvar ax].[7] •

5.1.2 Theorem [BL$^+$ compl lemma]:

Under the above assumptions 2 through 4 any globally valid BL$^+$ formula is provable using the indicated axiomatization (that is the BL$^-$ axiomatization together with [Tvar ax] and [V = thm]).

Proof:

We use the same approach as in the previous lemma. For a given globally valid formula P in BL$^+$ we derive a globally valid BL formula P' such that P' $\Rightarrow$ P is provable. P' is provable by the preceding lemma. Thus P is provable.

The construction of P' also proceeds roughly as before. We let Q be the formula P with Tv[x] substituted for each occurrence of <TV[x]>. Since P is valid it must hold for all states. Thus Q must hold no matter what functions we use for the Tv. We let P' be

$$\forall\, Tv_1\text{: func[var } T_1\text{] val } T_t \;\ldots\; \forall\, Tv_n\text{: func[var } T_n\text{] val } T_n \,.\, Q$$

We have just argued that P' is globally valid. We can prove P' $\Rightarrow$ P by defining the relations TR such that

$$x \text{ TR } y \quad \text{iff} \quad y = \langle TV[x] \rangle$$

---

[7]This is probably the most straight-forward proof of the lemma. However the quantification over functions is not essential. Since Tvar is just an isomorphism between L_T and the positive natural numbers (only equality must be preserved, since it is the only operation on l-values) we may take P' to be the formula with all instances of Tvar removed and l-value quantification replaced by quantification over integers. (In the model this corresponds to taking each L_T to be the nonnegative integers, and A_T the identity function.) [Tvar ax] suffices to first reintroduce the applications of Tvar (since they may only occur in equalities which are preserved) and then to replace quantification over integers by quantification over l-values where needed.

By [V = thm] the TR are functions. Thus we may substitute TR for each function TV and obtain the desired result. •

5.1.3 Theorem [BL^{++} compl lemma]:

Under assumptions 2 through 4 any globally valid BL^{++} formula is provable using the BL$^+$ axiomatization.

Proof:

We show that for a given a globally valid BL^{++} formula P, we can obtain a globally valid BL$^+$ formula P' such that given P' we can prove P using purely logical axioms and inferences. We then apply [BL$^+$ compl lemma] to obtain a proof for P'.

If a BL^{++} formula contains a term <NewT[]> then all Russell terms involving NewT must be identical to that term. Since P is valid it must hold for all states, and in particular for all values of s_a[T]. The terms <a> are the only ones which depend on this value. Thus if we obtain P' by replacing all occurrences of <NewT[1]> by a new (universally quantified) variable then P' must still be globally valid. We prove the implication P' $\Rightarrow$ P by resubstituting <NewT[1]> for this variable. •

The following lemma reduces the proof of the main theorem to a slightly simpler problem.

5.1.4 Theorem [cond RPL compl lemma]:

Assume that for every quantifier free (only universally quantified) RPL basic statement P we can find a BL^{++} formula P' such that P' $\Rightarrow$ P is provable and P' is equivalent to P whenever all Russell terms in P terminate. Then

[RPL rel compl thm] holds.

Proof:

Consider a quantifier free basic statement P which has the form

$$(\wedge_i \mathrm{Term}(a_i)) \wedge Q$$

where the $<a_i>$ are all the Russell terms appearing in P, and Term(a) is a BL^{++} formula which states that a terminates, for example:

$$\exists x . C_a(x)$$

By assumption we can find P' such that P' ⇒ P is provable. P ⇒ P' is true whenever the a_i terminate. It is now vacuously true when they don't. Thus P and P' are equivalent.

This argument easily generalizes to quantifier free basic statements P in disjunctive normal form in which each individual disjunct is preceded by the appropriate collection of Term($\cdot < i$) conjuncts. (Simply apply the preceding argument to each disjunct in turn.) We now eliminate the "quantifier free" restriction.

Assume that P' ⇒ P (with implicit universal quantification of any free variables) is provable. By predicate calculus rules so are

$$(\forall x{:}S . P') \Rightarrow (\forall x{:}S . P)$$

$$(\exists x{:}S . P') \Rightarrow (\exists x{:}S . P)$$

Now assume that given any quantifier free basic statement Q in the appropriate from (disjunctive normal form, each disjunct preceded by termination constraints) we can find an equivalent formula Q' such that Q' ⇒ Q is provable. Given an arbitrary formula P we first convert it to prenex normal form and then apply the given construction to the formula M we obtain by deleting the

quantifiers. This gives us a formula M' such that M' ⇒ M is provable. We then add the sequence of quantifiers appearing in P to M' to obtain the formula P'. By (induction on) the above observation P' ⇒ P is also provable.

We conclude by noting that if Q is a basic statement which insures termination we can obtain an equivalent formula P in the right format by converting it to prenex and disjunctive normal form and conjuncting the appropriate termination conditions onto each disjunct. P ⇒ Q is provable by predicate calculus rules. By the above argument we get a BL^{++} formula P' such that P' ⇒ P is provable in RPL. P' is equivalent to Q and therefore valid. Thus P' is provable by assumption 4. It follows that Q is provable. ●

5.2. Outline of the Main Proof

It will be convenient to prove something stronger than the hypothesis for the preceding lemma.

5.2.1 **Definition** [semi-basic def]:

A formula Q is *semi-basic* if all Russell terms have **val** or **var** signatures, and the formula has the form:

$$\forall f_1 \ldots \forall f_n . P_{f_1}(f_1) \wedge \ldots \wedge P_{f_n}(f_n) \Rightarrow P$$

where all the f_i have function signature, and where those identifiers in P that have func signatures, appear inside Russell terms, and are not bound in a surrounding let- or letrec-block, are among the f_i; and where for each i,

$$\exists f_i . P_{f_i}(f_i)$$

is provable. We refer to P as the body of the semi-basic formula.

5.2.2 Definition [compl det def]:

Let x have val or var signature. A formula P(x) *completely determines* x for a given state and environment if it is in $3L^{++}$ and

$$\forall x \forall y \,.\, (P(x) \wedge P(y)) \Rightarrow x = y$$

holds. Note that $C_{<a>}(x)$ always completely determines x. $P_{<a>}(x)$ completely determines x whenever a terminates.

We extend this to the case of an identifier f with function signature as follows. Let α be the general application $[\bar{w},\bar{y}]f[\bar{x}]$. Let ζ be the list of arguments with var signature. P(f) *completely determines* f if each $Q_{result}(x)$, $Q_{var,j}(x)$, and $Q_{New,T}(x)$ completely determines x whenever α terminates[8], and if P(f) has the form

$$\forall y_1 \ldots \forall y_k \forall x_1 \ldots \forall x_n \forall w_1 \ldots \forall w_m \,.\, Q_{result}(<\alpha>) \wedge$$
$$\wedge_j\, Q_{var,j}(<\alpha;\, V[z_j]>) \wedge$$
$$\wedge_T\, Q_{New,T}(<\alpha;\, NewT[]>)$$

Less formally, we require that P(f) completely determines both the result of an application of f and the resulting state. Again, the complete description of a function expression completely determines its argument. Even the partial description of a function *identifier* completely determines its argument. (Recall that identifiers by themselves always terminate.)

We can write P(f) in prenex normal form (after appropriately renaming the universally quantified variables) as

[8]That is, when $T(\alpha)$, as defined in the last chapter, holds.

$$Qu \,.\, \wedge_i\, R_i(E_i)$$

where Qu is a string of universal quantifiers and each $R_i(E_i)$ completely determines some extended general application of f. If P(f) completely determines f and Q(f) is

$$Qu \,.\, \wedge_i\, (S_i \Rightarrow R_i(E_i))$$

where each S_i is some condition on the universally quantified variables, then Q(f) completely determines f under conditions S_i.

We will show inductively that for any semi-basic, quantifier free, formula P we can find an equivalent (when all Russell terms terminate) formula P' such that its body is in BL^{++}. Furthermore we will preserve the following properties:

1. The formulas $P_{f_i}(f_i)$ in P' completely determine the f_i under conditions which are satisfied by all applications of f_i in Russell terms in the body of P'. That is, $P_{f_i}(f_i)$ completely determines any extended general application which corresponds to (i.e. the actual application takes place with the same arguments, argument values and allocation components as) an application of f_i in such a Russell term. (In almost all cases f_i will be completely determined under all conditions. The discussion of recursive declarations should make it clear exactly what the exceptions are.)
2. No applications of functions, other than those corresponding to the built-in side effect free operators in the Russell subset, are introduced into the body of the formula. This applies to applications both inside and outside of Russell terms.

The second property insures that if P is a basic statement then the function identifiers f_i described by the first part of P' will not appear in the body Q of P'.

Thus Q and P' are equivalent and $Q \Rightarrow P'$ is provable. Thus we obtain a formula Q *in* BL^{++} with the desired properties. [RPL rel compl thm] then holds by the preceding lemma.

We will first define the syntactic complexity of a programming language expression to be some non-negative integer. We then show that given any semi-basic RPL formula, we can simplify it to one which only simpler programming language expressions appearing in its body.

Given a semi-basic RPL formula Q its body is either already a BL^{++} formula, in which case we're done, or it contains Russell terms which do not have the form <V[x]> or <NewT[]>. We then chose a term <a> appearing in the body of P such that a has complexity m, and the body of Q contains no Russell terms of complexity greater then m. We then give a Q', with body in BL^{++}, that is equivalent to Q. We also give a proof for $Q' \Rightarrow Q$. We will use the induction hypothesis that globally valid formulas with at most n-1 occurrences of Russell terms of complexity m are provable.

Usually we do not explicitly reduce the body P of Q all the way to BL^{++}. Instead we reduce P to an equivalent formula P' which has at least one fewer Russell term of the highest complexity. We then show that the implication $P' \Rightarrow P$ is provable. We can then apply the induction hypothesis to get the rest of the way to BL^{++}. Only a few occasions do we need to either apply the induction hypothesis in a less direct way or refer to the function description part of Q rather than just its body.

We define the complexity of a programming language expression inductively:

1. The complexity of an identifier (or constant) is 1.

2. The complexity of a simple operator like '+', of an assignment, of a conditional, or of a **while'** loop is 2 plus the sum of the complexities of the subexpressions.
3. The complexity of V[a] is 2 plus the complexity of a, unless a is an identifier in which case it is 0.
4. The complexity of a sequence is 1 plus the complexity of the subexpressions.
5. The complexity of

   while c **do** a **od**

   is defined to be that of

   while' c **do** a **od**; c

   plus 1.
6. The complexity of NewT[k] is 0 if k is the constant '1' and 1 plus the complexity of the argument otherwise.
7. The complexity of a let- or letrec-block is 2 plus the maximum of
   a) the sum of the complexities of the subexpressions, and
   b) the maximum of the complexities of the extended general applications (cf. [compl descr def]) of the function expressions appearing on the right hand side of declarations.

   The latter clause insures that the complexity of a block is larger than those of Russell terms appearing in $P_{<f>}(f)$ where f is the right side of a function declaration.
8. The complexity of a function construction is 1 plus the complexity of the body.

9. The complexity of a function application with n arguments is 1 plus the complexity of the operator if all arguments are identifiers and 2n plus the complexities of all subexpressions if they are not. (This assures that an application of [arg ax] reduces the complexity of a formula.)

The above definition was contrived to insure that each of the "simplifications" we give in the following inductive proof actually reduce the complexity of the programming language expressions. It has no other significance.

At this point we consider the most complex Russell term <a> (or one such term if there are several) appearing in P, the body of the formula Q which we want to simplify. In general a will have the form

$$a_1; \dots ; a_n$$

To eliminate <a> we distinguish three cases.

If a_n is an identifier we let P', the simplified version of P, be P with <a> replaced by the identifier. [Id val ax] then allows us to prove $P' \Rightarrow P$.

If $n \neq 1$ and a_n is either NewT[] or TV[x] where x is a variable, we simplify a by applying the effect axiom corresponding to the outermost operator of a_{n-1}. We return to this case later.

The following section deals with the remaining case in which we can further simplify a_n.

5.3. Applying Value Axioms

We consider the case in which a_n is an expression other than an identifier, an application of TV to an identifier, or an application of NewT. We simplify a_n by

applying the value axiom associated with the outermost operator in a_n.

It is useful to abbreviate

$$a_1; \dots ; a_{n-1}$$

as α.

We will consider various possible outermost operators separately.

Simple operation -

A_n has the form

b op d

[op val ax] allows us to show that <b op d> is equal to

<b> op <b; d>

It follows that <a> is equivalent to

<α; b> op <α; b; d>

Thus we can prove that P is equal to P' which is obtained by substituting the above for <a>. Note that our definition of complexity was cleverly contrived so that <b; d> is less complex than <b op d>. Thus our induction hypothesis allows us to assume that P' can be reduced to BL^{++}.

V operation with complex argument -

A_n has the form

V[b]

where b is an expression other than a single identifier.

Let x be a new identifier. Let R be P with <a> replaced by <α; b; V[x]>. The meaning of V[b] is the same as that of b; V[x] if x is bound to <b>. Thus

$$x = <b> \;\Rightarrow\; R \qquad (P')$$

is equivalent to

$$x = <b> \;\Rightarrow\; P$$

which is equivalent to P. $P' \Rightarrow P$ is directly provable using [V arg ax]. Our measure of complexity was again contrived so that P' is simpler than P, that is ti contains one less Russell term of greatest complexity.

Assignment -

A_n has the form

$$b := d$$

Here we just substitute $<\alpha; b; d>$ for $<a>$. [:= val ax] allows us to prove that the result is equivalent to the original formula.

Conditional -

A_n has the form

if c **then** b **else** d **fi**

[if val ax] lets us prove that $<a_n>$ is equivalent to $<c; b>$ if $<c>$ holds, and $<c; d>$ otherwise. Thus if we let R be P with the conditional replaced by $<c; b>$ and R' be P with the conditional replaced by $<c; d>$ then P is provably equivalent to

$$(<c> \Rightarrow R) \wedge (\neg <c> \Rightarrow R')$$

Again the rewritten formula no longer has occurrences of $<a>$ and the newly introduced simple terms have complexity less than $<a>$.

Loop -

By [WH val ax] these can always be rewritten as **false**.

Variable allocation -

A_n has the form

NewT[k]

The case in which k is the constant 1 is of course treated separately. Let i be a new identifier. Let R be the formula P with $<a>$ replaced by

$$Tvar[i + <\alpha; k> - 1]$$

Let P' be

$$i = <\alpha; NewT[]> \;\Rightarrow\; R$$

The last part of [New ax] allows us to prove

$$i = <NewT[]> \;\Rightarrow\; Tvar[i + <k> - 1] = <NewT[k]>$$

By [U validity rule] this gives us

$$i = <\alpha; NewT[]> \;\Rightarrow\; Tvar[i + <\alpha; k> - 1] = <a>$$

This gives us

$$i = <\alpha; NewT[]> \;\Rightarrow\; R = P$$

Since i does not occur in P, it follows that P = P'.

Let block -

A_n has the form

let x == b **in** d **ni**

We may assume wolog that x does not occur elsewhere in P. (Otherwise use [let rename ax] to substitute a new variable for x in a_n.)

We first consider the case in which x does not have function signature. Let R be the formula P with the occurrence of a_n in $\langle a \rangle$ replaced by b; d. We then let P' be

$$\forall x . (x = \langle \alpha ; b \rangle) \Rightarrow R$$

P' has to be (globally) valid since the meaning of the block in an environment in which x is already bound to $\langle b \rangle$ is the same as that of $\langle b; d \rangle$. $P' \Rightarrow P$ is easily provable using [let ax].

To deal with the case in which x has function signature we first recall that we are actually simplifying a semi-basic formula Q of the form

$$\forall f_1 \ldots \forall f_n . P_{f_1}(f_1) \wedge \ldots \wedge P_{f_n}(f_n) \Rightarrow P$$

We replace the condition $x = \langle \alpha ; b \rangle$ in the above definition of P' by $P_{\langle b \rangle}(x)$ since the resulting formula would otherwise contain applications of x with no description of x among the P_{f_i}. Thus P' would not be semi-basic. Furthermore the condition itself introduces terms with function signature.

We let R' be

$$\forall f_1 \ldots \forall f_n \forall x . P_{f_1}(f_1) \wedge \ldots \wedge P_{f_n}(f_n) \wedge P_{\langle b \rangle}(x) \Rightarrow R$$

where R is defined as above. R' is clearly simpler than Q.

We show $R' \Rightarrow Q$ as follows. We take $x = \langle \alpha ; b \rangle$. By [fn val ax] this is equivalent to $x = \langle b \rangle$. By applying [fn subst ax] (possibly repeatedly) to each extended general application in $P_{\langle b \rangle}(x)$ and $P_{\langle b \rangle}(\langle b \rangle)$ we prove that the two are equivalent. $P_{\langle b \rangle}(\langle b \rangle)$ is semi-basic and thus provable by induction hypothesis. We now know that R' implies the following formula:

$$\forall f_1 \ldots \forall f_n \forall x . P_{f_1}(f_1) \wedge \ldots \wedge P_{f_n}(f_n) \Rightarrow (x = \langle \alpha ; b \rangle \Rightarrow R)$$

Q follows by [let ax].

Conversely, we know that $P_{\langle b \rangle}(x)$ holds then any terminating application of b is equivalent to the corresponding application of x. Thus no terminating Russell term can distinguish between a binding of $\langle b \rangle$ to x and an arbitrary binding satisfying $P_{\langle b \rangle}(x)$. It follows that $Q \Rightarrow R'$ whenever the block terminates.

Function constructions -

Terms with func signature are not allowed in this position since we are restricting ourselves to semi-basic formulae.

Function applications -

A_n has the form

$$f[b_1, \ldots, b_n]$$

If one of the b_i is not an identifier we can simplify a_n by applying [arg ax]. Let $\bar{x}$ be a sequence of n new identifiers. Recall that $\bar{x} = \bar{b}$ abbreviates

$$x_1 = \langle b_1 \rangle$$

$$x_2 = \langle b_1; b_2 \rangle$$

$$\vdots$$

$$x_n = \langle b_1; b_2; \ldots; b_n \rangle$$

Let R be P with $\langle a \rangle$ replaced by

$$\langle \alpha ; b_1; \ldots; b_n; f[\bar{x}] \rangle$$

Let R' be

$$<\alpha>(\bar{x} = \bar{b}) \Rightarrow R$$

If we eliminate abbreviations this expands to

$$x_1 = <\alpha; b_1> \wedge \ldots \wedge x_n = <\alpha; b_t; \ldots; b_n> \Rightarrow R$$

We eliminate equalities involving function identifiers by replacing them with a partial description, exactly as we did with simple declarations.

We now examine the operator f. If it is a sequence it can be easily simplified using [fn val ax]. If it is a conditional or a let- or letrec-block, we simplify it in basically the same way as we simplified the corresponding construct with val or var signature. The same applies if it is an application with arguments which are not identifiers. In the latter three cases we can easily obtain a semi-basic P' equivalent to P (in the event of termination) of the form

$$S \Rightarrow P_r$$

where P_r denotes P with $<a>$ replaced by $<\alpha; f'[\bar{b}]>$, f' is simpler than f, and P' $\Rightarrow$ P is provable using the same arguments as for the val or var case, together with [fn subst ax]. In the case of the conditional we get a disjunction of several such formulae, which we then simplify further individually. In the case of an application with an operator which has one of these forms we apply the same reduction to its operator.

This leaves us with the case in which a_n consists of a (possibly repeated) application of a function identifier or construction to identifier arguments.

An application of a function construction can be simplified by using [func rename ax] to rename the arguments to match those of the parameters, and then using [η axiom] to eliminate both the construction and the innermost

application.

To simplify an application of an identifier f, we observe that one of the hypotheses in Q must be a formula $P_f(f)$ which completely determines f. One conjunct in $P_f(f)$ describes the result of the repeated application of f. It has the form[9]

$$\forall\bar{w}\,\forall\bar{y}\,\forall\bar{x}^{\,j} \ldots \forall\bar{x}^{\,t}\,\forall\bar{x}\; Q_{RES}(F)$$

where F is the Russell term

$$<z_1 := w_t; \ldots; z_m := w_m; NewT_1[y_t]; \ldots; NewT_k[y_k]; f[\bar{x}^{\,j}] \ldots [\bar{x}^{\,1}]\,[\bar{x}]>$$

and $\bar{z}$ is the sublist of $\bar{x}$ with var signature. (None of the identifiers in $\bar{x}^{\,i}$ have var signature.)

Assume wolog that the identifiers $\bar{x}$ and $\bar{x}^{\,i}$ are those that occur in the actual application we are trying to simplify. Let r be a new identifier. We then let P' be

$$\begin{aligned}\forall\bar{w}\,\forall\bar{y}\,\forall r\,.\,\textstyle\bigwedge_i <NewT_i[y_i]; NewT_i[]> &= <\alpha; NewT_i[]>\\ \wedge \textstyle\bigwedge_i w_i &= <\alpha; V[z_i]>\\ \wedge\ P_{RES}(r)&\\ \Rightarrow R&\end{aligned}$$

where R is P with $<a>$ replaced by r. P and P' are equivalent when $<a>$ terminates since the conditions on $\bar{w}$ and $\bar{y}$ insure (by [s compactness thm]) that the applications in F and $<a>$ produce the same result. Since P_{RES} completely

---

[9]In general the Russell term may be a bit more complicated. It may be a general application of a general application etc. Since all but the outermost application produce results with func signature we can use [fn val ax] and [fn subst ax] to simplify it to the form we give.

determines r, we know that $r = F$ and therefore $r = \langle a \rangle$.

Given that $P_{RES}(F)$ and P' hold we can easily prove P. We first have to prove that α and the sequence of assignments an allocations in F result in the same value in the z_i and the same number of allocated locations. The provability of this statement follows from the induction hypothesis. We then use [appl state ax] to prove that $F = \langle a \rangle$. Thus $P_{RES}(\langle a \rangle)$ holds. We take $r = \langle a \rangle$ to get our desired conclusion.

Recursive declarations -

This is virtually identical to the treatment of recursive declarations in the next section.

5.4. Applying effect axioms

We return to the case in which a_n is either V[x] or NewT[]. To simplify a in this case, we generally apply the effect axiom or rule associated with the outermost operator or construct in a_{n-1}. We consider the various possibilities separately. It will be useful to abbreviate

$$a_1; a_2; \ldots; a_{n-2}$$

as β.

Identifier -

A_{n-1} may be a constant or variable. In this case [id ef ax] allows us to replace $\langle a \rangle$ by

$$\langle \beta; a_n \rangle$$

Simple operation -

A_{n-1} may have the form

$$b \text{ op } d$$

We can rewrite $\langle a \rangle$ as

$$\langle \beta; b; d; a_n \rangle$$

using [op ef ax].

V operation -

A_{n-1} may have the form V[b]. If b is an identifier, [V ef ax] and [id ef ax] together allow us to delete it from $\langle a \rangle$. If b is anything else it suffices to use [V ef ax] to replace V[b] by b. (Note that in the first case our definition of complexity forced us to combine the two steps. This could have been avoided with a different and more complex definition.)

Assignment -

Assume a_{n-1} has the form

$$b := d$$

First assume a_n is V[x]. Let R be P with $\langle a \rangle$ replaced by

$$\langle \beta; b; d \rangle$$

and let R' be P with $\langle a \rangle$ replaced by

$$\langle \beta; b; d; V[x] \rangle$$

If $\langle b \rangle = x$ then P is equivalent to R, otherwise to R'. (Again recall our assumption about complete aliasing). Thus if x and b have the same signature we let P' be

$$(\langle b\rangle = x \Rightarrow R) \wedge (\langle b\rangle \neq x \Rightarrow R')$$

We use P' = R' if x and b have different signatures. In the latter case [:= ef ax] immediately gives us P = P'. In the former case it yields

$$((\langle b\rangle = x) \Rightarrow (R = P)) \wedge ((\langle b\rangle \neq x) \Rightarrow (R' = P))$$

and thus

$$((\langle b\rangle = x) \Rightarrow (P' = P)) \wedge ((\langle b\rangle \neq x) \Rightarrow (P' = P))$$

or just

$$P = P'$$

The remaining possibility is that a_n is NewT[]. In this case we let P' be P with <a> replaced by

$$\langle\, b; d; \mathrm{NewT}[]\rangle$$

The third part of [:= ef ax] then allows us to prove P = P'.

Conditional -

This is almost identical to the case of the conditional in the last position. We again let P be quantifier free. If a_{n-1} has the form

if c **then** b **else** d **fi**

we take Q to be P with <a> replaced by

$$\langle\, c; b; a_n\rangle$$

and Q' to be P with <a> replaced by

$$\langle\, c; d; a_n\rangle$$

We then take P' to be

$$(\langle c\rangle \Rightarrow Q) \wedge (\neg \langle c\rangle \Rightarrow Q')$$

Loop -

A_{n-1} may be of the form

while c **do** b **od**

or of the form

while' c **do** b **od**

We abbreviate the latter as WH'.

In the first case we use [WH ef rule] to rewrite the **while**-loop into a **while'**-loop. Thus we assume we are in the second case.

We will find a P' equivalent to P when the loop (including the final evaluation of the condition) terminates, and such that P' and [WH' ef rule] we can deduce D.

In order to apply [WH' ef rule] we need to pick an ordering $\subset$, a constant k, and a variant function v. We chose v to be the whole state of the computation. That is, we let v be

$$(\langle V[x_1]\rangle, \ldots, \langle V[x_j]\rangle, \langle \mathrm{NewT}_1\rangle, \ldots, \langle \mathrm{NewT}_k\rangle)$$

where the x_i are all variables occurring in <a>, and the T_i are all types. The idea then is to let $\subset$ be something like the transitive closure of the transition function associated with one iteration of the loop.

Thus we let S^* be the reflexive transitive closure of the following relation S on

$$\{\, (a_1, \ldots, a_j, l_1, \ldots, l_k) \,\} \cup \{\, \infty \,\}$$

Let xSy hold iff neither x nor y are ∞ or

$$(v = y) \Rightarrow \text{not} <c> \text{ and } (<c; b> v = x)$$ [10]

S in general is not well founded. Thus we have to define the relation $\subseteq$ to be S restricted to those states in which the loop terminates. More formally let $x \subseteq y$ hold iff xS^*y and there is only a finite collection of values z such that zS^*y holds, or if there is only a finite collection of z such that zS^*x holds and $y = \infty$.

As expected we let $k = \infty$. The statement $s \subseteq k$ is equivalent to saying that s leads to termination. $x \subseteq y$ where both x and y are (j+k)-tuples defining a state is equivalent to the statement that both x and y lead to termination and that if the while'-loop is started in state y then after some number of iterations of the loop state x will be reached. Thus $\subset$ is clearly well-founded. Furthermore this statement is expressible in BL^{++} and therefore provable. By construction the other hypothesis of the [WH' ef rule] is also satisfied. Since it is less complex than P it is provable by the induction hypothesis.

We introduce new identifiers w_i which represent the value of the variables x_i at the end of the loop. Similarly we let z_{T_i} represent the value of $NewT_i[]$ at the end of the loop. We define a formula R to be P with <a> replaced by the appropriate w_i or z_i, depending on a_n.

[10]Formally we need to argue that this relation is expressible in RPL. We can rewrite the above equation in BL^{++} using complete descriptions of the Russell terms. The resulting formula will be equivalent for states (and environments) in which c; b terminates. In other states it will be vacuously true. Thus the equivalence will be true, and thus provable by induction hypothesis, whenever $v \subset \infty$. ($\subset$ is defined in the next paragraph. That definition and the fact that xSy whenever c; b doesn't terminate in state y insure that $v \subset \infty$ only if c; b terminates.) This will suffice. We then rewrite the formula in BL^{*} using the procedures given in [BL compl lemma], [BL^{+} compl lemma], and [BL^{++} compl lemma]. Our assumptions guarantee that a relation given by such a binary predicate is then definable.

We let P' be

$$v \subset k \wedge \forall z\, (((v' \subseteq v) \wedge \neg c') \Rightarrow R)$$

Here v' and c' represent v and c respectively with each $V[x_i]$ replaced by w_i and each $NewT_i[]$ replaced by z_{T_i}. In the case of c we first rewrite it in BL^{++} using its complete description[11].

It should be observed that

$$v' \subseteq v \wedge \neg <c'>$$

Is just a formal way of saying that v' is the final state produced by the while'-loop when started in state v. (There is after all just one state which is reachable from v by some number of loop iterations in which c is false.) Thus P' is true whenever the loop terminates and R holds when all the w_i are the final values of x_i, and the z_{T_i} are the values given by $<NewT_i[]>$. It follows from the definition of R that P' is exactly the formula we promised.

$P' \Rightarrow P$ is easily provable. We know that the two hypotheses of [WH' ef rule] hold. Since we are given $v \subset k$ as well, we conclude that

$$(<WH'> v) \subseteq v \wedge \neg <WH'> <c>$$

We obtain the desired result by substituting $<WH'>\ x_i$ for each x_i' in P' and then observing that this transforms R back into P[12]. Thus D holds.

[11]More precisely we replace $\neg c$ by $\forall x\, .\, C_{<c>}(x) \Rightarrow \neg x$
This isn't quite equivalent when c doesn't terminate, but we are not interested in that case. We argue in a later footnote that the proof of $P' \Rightarrow P$ still goes through in spite of this change.

[12] [WH' ef rule] allows us to conclude that c as written evaluates to false at the end of the loop. We can show that this means the BL^{++} version also evaluates to false since

$$\neg c \Rightarrow (C_{<c>}(x) \Rightarrow \neg x),$$

or

$$c \vee \neg C_{<c>}(x' \vee \neg x$$

Variable allocation -

A_{n-1} has the form

$$\text{New}\ T[k]$$

If a_n is either V[x] or NewT'[], where T' and T are different, we can use the first two parts of [New ax] to replace $<a>$ by $<\beta; a_n>$.

The remaining case is very similar to the treatment of NewT in the previous section.

Let i be a new identifier. Let R be the formula P with $<a>$ replaced by

$$\text{Tvar}[i + <\beta; k>]$$

Let P' be

$$i = <\beta; \text{NewT}[]> \Rightarrow R$$

The first part of [New ax (3)] allows us to prove

$$i = <\text{NewT}[]> \Rightarrow \text{Tvar}[i + <k>] = <\text{NewT}[k]; \text{NewT}[]>$$

By [U validity rule] this gives us

$$i = <\beta; \text{NewT}[]> \Rightarrow \text{Tvar}[i + <k> - 1] = <a>$$

This gives us

$$i = <\beta; \text{NewT}[]> \Rightarrow R = P$$

It follows that P = P'.

Let block -

This is again essentially the same treatment as in the preceding section. A_{n-1} has

is valid, insures termination ($C_{<c>}(x)$ is false if c doesn't terminate), and is therefore provable by induction hypothesis.

the form

$$\text{let } x == b \text{ in } d \text{ ni}$$

Assume that P has no explicit quantifiers and use [let rename ax] to insure that x does not occur elsewhere in P. We let R be the formula P with the occurrence of a_{n-1} in $<a>$ replaced by $<b; d>$. We then let P' be

$$x = <\beta; b> \Rightarrow R$$

P' ⇒ P can be proved using [let ax]. We deal with function declarations as in the previous section.

Function constructions -

If a_{n-1} has the form

$$\text{func}[\boxed{x}]\ \{a\}$$

it can be deleted using [func ef ax].

Function applications -

The same argument as in the previous section applies here, except that a different conjunct of the partial function description is used.

Recursive declarations[13] -

A_{n-1} has the form

$$\text{letrec } \bar{x} == \bar{b} \text{ in } d \text{ ni}$$

We assume that a_n is V[z]. The other case is similar.

The following argument needs to be modified in an obvious way if this simplification is being carried out as part of the simplification of an application.

[13] The reader may wish to replace this subsection by the statement that "This is virtually identical to the treatment of recursive declarations in the previous section".

In that case we will need to discuss termination of the (possibly repeated) application of the body, rather than simply termination of the body. The reader should keep in mind that we are not using the full version of [letrec ax] because we are dealing with the situation outside any application.

Our approach is very similar to that for a non-recursive function declarations. We let R be P with <a> replaced by

$$<\beta;\ d;\ V[z]>$$

We again add partial descriptions of the x_i as hypotheses to Q, and replace the body P by R. Note that the partial descriptions $P_{x_i}(x_i)$ exist since they are the partial descriptions of the Russell terms

$$<\text{letrec } \bar{x} == \bar{b} \text{ in } x_i \text{ nl}>$$

We effectively replace P by the formula R'

$$\forall\bar{x}\,.\,P_{x_1}(x_1) \wedge \ldots \wedge P_{x_n}(x_n) \Rightarrow R$$

R' and P are equivalent by the same argument as for non-recursive declarations.

It is somewhat more difficult to prove that $R' \Rightarrow P$. By [$\boxed{\exists}$ ~ ax] we can chose $\bar{x}$ to satisfy

$$\bar{x} \sim \bar{b}$$

Using [letrec ax] we can immediately show that $R \Rightarrow P$. The only remaining problem is to show that the $P_{x_i}(x_i)$ are satisfied.

We first order the extended general applications by their calling relationship. That is, we write

$$(E_i, \bar{z}_i, \bar{w}_i, \bar{y}_i) \subset (E_j, \bar{z}_j, \bar{w}_j, \bar{y}_j)$$

if the j^{th} extended general application with arguments (possibly for repeated applications) $\bar{z}_j$, argument values $\bar{w}_j$ and allocation components $\bar{y}_j$ results in a call to the i^{th} general application with the indicated arguments and state. As with the loop, we turn the relation into a well-founded one by explicitly excluding nonterminating chains.

We proceed to prove inductively that that the $\bar{x}$ satisfy the complete description S_j of the extended general application $(E_j, \bar{z}_j, \bar{w}_j, \bar{y}_j)$. We have to prove as the induction step(s) that

$$\forall\bar{x}\,\forall\bar{z}\,\forall\bar{w}\,\forall\bar{y}.\ P'_{x_1}(x_1) \wedge \ldots \wedge P'_{x_n}(x_n) \Rightarrow (\bar{x} \sim \bar{b} \Rightarrow S_j)$$

where $P'_{x_i}(x_i)$ is $P_{x_i}(x_i)$ with the added conditions that the extended general applications in question be less than (by the $\subset$ ordering) $(E_j, \bar{z}_j, \bar{w}_j, \bar{y}_j)$. Using [$\sim \Rightarrow \sim$ ax] the above reduces to

$$\forall\bar{x}\,\forall\bar{z}\,\forall\bar{w}\,\forall\bar{y}.\ P'_{x_1}(x_1) \wedge \ldots \wedge P'_{x_n}(x_n) \Rightarrow ((\wedge_i\, x_i \sim b_i) \Rightarrow S_j)$$

This can be derived from the induction hypothesis by showing

$$\forall\bar{x}\,\forall\bar{z}\,\forall\bar{w}\,\forall\bar{y}.\ P'_{x_1}(x_1) \wedge \ldots \wedge P'_{x_n}(x_n) \Rightarrow S'_j) \wedge T(E'_j)$$

where S'_j is S_j with the application of x_k replaced by b_k, and E'_j is the similarly modified general application. We then use [~ def] to get back to the preceding formula. ●

CHAPTER 6

EXTENSIONS AND CONCLUSIONS

We begin this last chapter by (somewhat informally) extending the logic presented so far to some other constructs. Most of these constructs are present in the full Russell programming language in some form. We discuss, in order, nondeterministic constructs, structured data objects in our treatment of variables, and finally the Russell notion of types as collections of functions. We then draw some general conclusions and discuss the motivations for and the results of our unusual treatment of nontermination.

6.1. The Role of Nondeterminism

In developing the logic we chose a completely deterministic language. There is good reason to do so. Since we treat Russell terms as a special kind of mathematical term they should behave like mathematical terms. In particular we want the equality

<a> = <a>

always to hold. If a is a nondeterministic programming language expression and <a> denotes the single value it produces this clearly no longer holds.

We might attempt to solve this by redefining <a> to denote the set of possible values returned by a. We would then extend the basic functions built into the logic so that when they are given sets of values by such a term they produce the corresponding sets of possible results. But this does not suffice.

Consider a programming language construct a OR b which non-deterministically evaluates either a or b. Now consider the term

<1 OR 2> + <1 OR 2>

Is the resulting collection of values {2, 4}, or is 3 a possible value as well? Clearly the answer depends on whether the two subexpressions refer to the same execution of 1 OR 2. Thus we would have to find some means of indicating this explicitly.

Thus we are lead to the following general approach: We can extend the notion of a Russell term to be the following:

label:< Russell_expression>

Here the label is used to identify the particular execution of the construct. In general the label will have to be structured in a way that corresponds to the construct itself. For example

x;y : <a; b>

refers to the value of the y execution of b following the x execution of a. This approach, although it is general, is clearly lacking in both elegance and simplicity. Thus we do not pursue this approach in detail, but rather resign ourselves to the fact that general nondeterminism is not a concept that is easily formalized using this approach. This in itself is an interesting contrast to Hoare logic.

It is possible to formalize a more restricted kind of nondeterminism, such as that described in [Boe 80]. There the order of argument and guarded command evaluation is nondeterministic, with the proviso that whatever order is chosen be chosen consistently. We refer to this as compile-time nondeterminism.

As an illustration consider a compile-time nondeterministic version of the above OR construct. We will refer to it as **OR**. Thus for particular a and b the semantics of a **OR** b is either that of a or that of b, and it is so independently of the context in which a OR b occurs. In this way we have explictly dictated that

$$\langle a\rangle = \langle a\rangle$$

even in the presence of "nondeterminism".

A construct of this type is easily axiomatized. A first attempt might be

[OR val ax]:

$$\langle a \text{ OR } b\rangle = \langle a\rangle \ \lor\ \langle a \text{ OR } b\rangle = \langle b\rangle$$

[OR ef ax]:

$$\langle a \text{ OR } b\rangle t = \langle a\rangle t \ \lor\ \langle a \text{ OR } b\rangle t = \langle b\rangle t$$

This is clearly a sound axiomatization, but again it doesn't quite suffice. The problem arises in trying to prove for example

$$\langle x := 1 \text{ OR } x := 2\rangle = \langle x := 1 \text{ OR } x := 2\rangle x$$

The above axioms do not state in any way that there is a relationship between which value and which state is produced by the OR construct. Thus the two axioms need to be combined to the following one:

[OR ax]:

$$\begin{array}{ll} & \langle a \text{ OR } b\rangle = \langle a\rangle \ \land\ \langle a \text{ OR } b\rangle t = \langle a\rangle t \\ \lor & \langle a \text{ OR } b\rangle = \langle b\rangle \ \land\ \langle a \text{ OR } b\rangle t = \langle b\rangle t \end{array}$$

or, using the abbreviation introduced in chapter 4,

$$\langle a \text{ OR } b\rangle \equiv \langle a\rangle \ \lor\ \langle a \text{ OR } b\rangle \equiv \langle b\rangle$$

The relative completeness proof can easily be generalized to this construct. We can rewrite a formula P involving OR into $Q \lor Q'$ where Q and Q' are obtained by substituting the first and second argument respectively for the OR construct.

We can easily treat a nondeterministic argument evaluation order in the same manner. Let OP be a binary operator corresponding to logical operator op, but with unspecified evaluation order subject to the same constraint as OR. We may write:

[unspec OP ax]:

$$\begin{array}{ll} & \langle a \text{ OP } b\rangle = \langle a\rangle \text{ op } \langle a; b\rangle \ \land\ \langle a \text{ OP } b\rangle t = \langle a; b\rangle t \\ \lor & \langle a \text{ OR } b\rangle = \langle b; a\rangle \text{ op } \langle b\rangle \ \land\ \langle a \text{ OP } b\rangle t = \langle b; a\rangle t \end{array}$$

The modification to the relative completeness proof is similar to that for the OR construct.

If we wanted, we could similarly reintroduce the compile-time nondeterministic guarded commands of [Boe 80].

In the last two cases it can reasonably be argued that this notion of nondeterminism is "the" correct one anyway. It seems counterintuitive, at least in this context, not to preserve the property that $\langle a\rangle = \langle a\rangle$. In the context of a type checking system similar to the one in Russell it is in fact intolerable, at least for type valued expressions. Furthermore language implementations are likely to preserve this property. It is perfectly acceptable for a compiler to reorder subexpression evaluation to improve efficiency, provided it does so based purely on the structure of the subexpression, and not on context.

Unfortunately, this approach is completely inadequate in the context of concurrent programming.

6.2. Structured Data Objects

One of the virtues of the programming logic presented here is its ability to express statements about structured data objects without significant additions to the logic. We can axiomatize arrays without talking about array updating in the logic or modelling arrays as functions. Instead we deal with them directly as sequences of l-values. This is both closer to most people's intuition and avoids some formal problems. (See for example the discussion of call by reference in [Gri 81]. It is hard to even state what call by reference means if we model arrays purely as functions.)

To illustrate, we consider the type T^n of arrays of T indexed by the integers 0 .. n-1. We need to look at only one new operation, namely the subscription operator ι. Informally $A \iota i$ where A has signature T' and i has signature Integer produces a *variable* which is the i^{th} component of the array A.[1]

The only serious difficulty introduced by this is that our assumption of no partial aliasing no longer holds. This has two consequences. First our model of l-values needs to be revised. We may take l-values for T^n to be sequences of T l-values of length n. $NewT^*$ yields such a sequence.

The second consequence is that our previous proof rules for assignment and storage allocation must be generalized. It is no longer the case that assigning to one l-value (e.g. an array component) does not affect a different l-value (e.g. the array). Thus we need a notion of disjoint l-values. We use the symbol $\diamond$ to denote this property.[2] Thus if we adopt the above notion of structured l-values we can formally state:

---

[1]Since ι is assumed to be the only subscription operator, array values are clumsy to use. To obtain an individual element of an array value one would have to assign the array to a new array variable and then apply the subscription operator to the variable. This is not really acceptable, but it suffices as an illustration. A more reasonable view of arrays is given in [Boe 80].

[2]Since $\diamond$ may take arguments of different signatures it has to be treated as a different logical primitive, rather than just a new function constant. If we used the full Russell signature calculus it could be treated either way.

6.2.1 Definition [$\diamond$ def]:

If l_1 and l_2 are two l-values then $l_1 \diamond l_2$ iff the following three conditions hold:

1. l_1 and l_2 are not equal.

2. If l_1 is structured, that is, is a set of basic l-values, then for each $l \in l_1$ $l \diamond l_2$.

3. If l_2 is structured, then for each $l \in l_2$ $l \diamond l_1$.

The following property is an immediate consequence of the definition:

[$\diamond$ ax]:

$$t_t \diamond t_2 \;\Rightarrow\; t_t \neq t_2$$

We can now generalize some of the chapter 4 axioms. One approach to this is to first strengthen the preceding axiom in the case in which the signatures of t_1 and t_2 mention only basic types. In this case we have

$t_1 \diamond t_2$	if the signatures are the same
$t_t \diamond t_2 \;=\; t_1 \neq t_2$	if the signatures differ

We then change [New ax] primarily in that we define NewT for composite types in terms of that for more basic types. Thus the second line of [New ax]

$$<NewT[k];\ NewT'[]> \;=\; <NewT'[]>$$

needs to be restricted to the case in which T and T' are different *and neither is composite*. We can then define $NewT^n$ as follows:

[$NewT^n$ ax]:

$$<NewT^n[k];\ a> \;=\; <NewT[(n+1)*k];\ a>$$

$$<NewT^n[k] \iota i> \;=\; <NewT[(n+1)*(k-1)+i-1]>$$

This is again much too specific for more practical purposes, so an adaptation of the Is_allocated approach might be preferable. The main change here is that

$$\text{Is\_allocated}[x] \Rightarrow \langle \text{NewT}[] \rangle \neq x$$

becomes

$$\text{Is\_allocated}[x] \Rightarrow \langle \text{NewT}[] \rangle \diamond x$$

The assignment axioms become

[rev := ef ax]:

$$\begin{array}{l} \langle a \rangle = x \;\Rightarrow\; \langle a := b; V[x] \rangle = \langle a; b \rangle \\ \langle a \rangle \diamond x \;\Rightarrow\; \langle a := b; V[x] \rangle = \langle a; b; V[x] \rangle \end{array}$$

No further changes are required. (The part of the axiom specifying the result of NewT[] after the assignment does not need to be changed.)

If no structured variables are involved then either of the above rules can always be applied. If arrays are involved some other axioms are clearly necessary to allow us to determine which case we're in. The following two axioms characterize location equality (aliasing) between elements of two arrays, and within the same array:

[array = ax]:

$$\begin{array}{lcl} \langle a \rangle = \langle b \rangle & = & (\forall i \,.\, \langle a{\downarrow}i \rangle = \langle b{\downarrow}i \rangle) \\ \langle a; j \rangle = \langle a; k \rangle & = & \langle a{\downarrow}j \rangle = \langle a{\downarrow}k \rangle \end{array}$$

Here j and k are expressions such that $\langle a; j \rangle$ and $\langle a; k \rangle$ have values between 0 and n-1. The universal quantifier ranges over 0 through n-1.

We can similarly describe the $\diamond$ relation:

[array $\diamond$ ax]:

$$\begin{array}{l} \langle a \rangle \diamond \langle d \rangle \;=\; (\forall i \,.\, \langle a{\downarrow}i \rangle \diamond \langle d \rangle) \\ \quad \langle a{\downarrow}i \rangle \diamond \langle a{\downarrow}j \rangle \;\vee\; \langle a{\downarrow}i \rangle = \langle a{\downarrow}j \rangle \end{array}$$

One further axiom is necessary[3]. Currently, if we assign to an array variable, we are only allowed to make conclusions about the value of the whole array. Thus we can conclude that after the array assignment a := b we have V[a] = V[b], but not that $V[a{\downarrow}1] = V[b{\downarrow}1]$. Thus we need to relate array values with component values. We formally state this as

[array V = ax]:

$$\langle V[a] \rangle = \langle V[b] \rangle \;=\; (\forall i \,.\, \langle V[a{\downarrow}i] \rangle = \langle V[b{\downarrow}i] \rangle)$$

The quantification is again intended to be over 0 .. n-1.

6.3. The Russell Notion of a Type

We outline the changes that must be made to the preceding development to accomodate the view of types introduced in [Dem 80] and [Boe 80]. Since the changes are substantial, we do not give many details. This section should be read as a plausibility argument rather than as a thorough development.

We treat data types simply as collections of operations (that is functions) that may be applied to, and thus impose an interpretation on, some universal set of values[4]. Since we have already dealt with functions as data objects, it is easy to conceive of manipulating types, that is collections of them, as data objects as well. In that sense we are not adding anything fundamental. Nonetheless a number of

[3]Another minor complication is that the notion of a general application needs to be slightly refined in the presence of overlapping variables. But this affects only the application of the formal system; the system itself remains valid.

[4]In a very operational view, we can regard these values as simply strings of bits. Note that conventional programming language implementations do implement everything, including function objects, as bit strings. More elegant mathematical models are suggested below.

detailed modifications are needed. They are listed below.

1. The definition of the set D of data objects is completely inadequate for the language described in [Boe 80].

   The **Image** type constructor[5] in that language explicitly allows us to introduce reflexive types. That is, we can define a type T such that **val** T and

   func[**val** T]**val** T

   are "isomorphic". As a consequence we can translate the untyped lambda calculus into Russell.

   We cannot define $D_{\mathbf{val}\ T}$ for types T such as the one above as easily as the constructions in chapter 2. This problem can easily be solved by using Scott's reflexive domain constructions. (See for example [Sto 77].)

2. In [Boe 80] there are no built-in operations of the kind we have been discussing. The '+' operation for integers does not exist by itself; instead there is a built-in type *Integer* which includes becomes

   $$<a\ \mathit{Integer}\$+\ b>^{6} \;=\; <a> + <a; b>$$

   Similar changes must be made for other operations. Assignment and TV are also associated with individual types. Thus we would restrict these axioms to those instances of assignment and TV that are associated with built-in types[7].

[5]This has been replaced by the more general product constructor **prod** in more recent versions of the Russell language.

[6]Abbreviations are provided so that this is normally not written in full. We express axioms in terms of the unabbreviated notation. The same comment applies to the TV[...] or V[...] notation we have been using throughout.

[7]In all cases, including the *Integer*$+ example, we have to be careful to apply these axioms only when the type name, e.g. *Integer*, really refers to the builtin type, and not a user-defined type which happens to share the name. This can be enforced by restricting the axioms for let, letrec, and func to apply only when no built-in types are redeclared. This forces the appropriate renaming to be done first.

   (One would hope that user-implemented instances of these operations satisfy the same axioms, but we certainly can't guarantee that.)

3. We must axiomatize any type constructions we introduce. Rules are needed to describe the individual functions which are components of the type values produced by constructions. For the language described in [Boe 80] this collection of rules would be fairly large. Later versions of the language make more use of general Cartesian product construction[8] to build types (i.e. Cartesian products of functions). Thus many of these rules would be needed independent of type manipulation.

   We further need a rule analogous to [fn subst ax] or the first part of [array = ax]:

   $$<T_t> \;=\; <T_2> \;\Rightarrow\; <T_t\$f> \;=\; <T_2\$f>$$

   This is necessary to infer any properties of declared type identifiers.

4. We must of course change the notion of a signature in the logic to correspond to that in the programming language. Equality of l-values in particular must be treated in a more general way since it is now possible for variables with different signature to alias[9] (or of course overlap). As in the preceding section, we need to improve our notion of a general application if we wish to use it.

[8]These can be viewed as records which prohibit modification of individual fields.

[9]Consider **func**[x: **val** *Integer*; y: **val** T; T: ...] { ... } [x, x, *Integer*]. This is a signature correct application, x and y have signatures **val** Integer and **val** T respectively inside the body of the function, but nonetheless they alias.

6.4. Conclusions

The logic we have presented gives us rules for reasoning about programs utilizing language constructs which have frequently been considered detrimental to such a task. Most of these rules have been exceptionally simple, even, in cases such as the assignment, where they represent substantial generalizations over Hoare's rules. Only our treatment of functions has been somewhat involved, but certainly no more so that other approaches of comparable generality.

Our philosophy has usually been to devise a set of proof rules which are as simple as possible, but sufficiently general so that we can derive more easily applicable rules from them. Thus it is somewhat inconvenient to carry out proofs in full detail using only the basic rules. But in return we have gained in terms of simplicity (except where the goal of relative completeness got in the way — see the next section).

The formal system interacts nicely with conventional informal reasoning about programs. In particular we have defined the meaning of Hoare style assertions and all our proof rules are generalizations of Hoare's. Thus if a section of the program is amenable to proof using Hoare's system we can easily translate such a proof into the present formalism. It is still possible to present proof outlines in the form of a program annotated with assertions, although this becomes less and less informative as one's programming style becomes more and more applicative.

It may be useful to develop some notational convention for inserting pre- and postconditions for arbitrary sub-expressions (rather than just at a ';'). We may also want to establish a syntactic shorthand for referring to the value yielded by the last expression. It would then be possible to derive rules similar to those in [Kow 77].

Our approach can be viewed as a combination of the following three ideas. We distinguish strictly between programming language and logic operations. This may be useful in general, and is virtually unavoidable in this context since it is not clear what the meaning of side effects in logical formulae is.

Programming logics such as Dynamic Logic or Constable's Programming Logic allow one to talk about the truth or falsity of a predicate after execution of some sequence of statements. The present logic generalizes this to the value of an arbitrary expression after a number of state-changing operations.[10] This allows us to obtain our simplified assignment axiom which does not explicitly deal with substitution, but instead relies on the ability to substitute equal terms in the predicate calculus. By thus reducing the problem to the derivation of equalities between certain simple kinds of terms, we avoid the pitfalls we might otherwise have encountered in dealing with aliased variables. We no longer have to worry about the ill-defined concept of substituting for "all aliases of a given variable".[11]

In order to express properties dealing with aliasing as easily as possible we have differed from the conventional approach in yet a third way. We consider l-values to be objects in their own right, rather than just synonyms for the value stored in them. This is certainly not very original; the two concepts have traditionally been distinguished both in informal and in formal denotational definitions of programming languages. It has been felt in the past that this was too "grubby" a concept to address explicitly with an axiomatization. The alternative however seems to introduce much

[10] This concept is expressible in Dynamic Logic and similar logics. It is already present in [Mir 71]. The difference is that we exploit that fact by using it as the basic primitive with which we express our axioms.

[11] This is clearly not a syntactic notion. Consider substituting for aliases of x in

<let y == x In V[y] nl> = 0

more complexity. It would no longer be possible to treat aliasing as simply the equivalence of two l-values. It would also have been necessary to treat structured objects differently; most conventional approaches introduce the applicative version of the data structure into the logic. Thus if we wanted to talk about arrays we would need to introduce a notion of an array update function into the logic.

It is also this very consistent distinction between l-values and r-values, together with the simple notion of declaration present in Russell (among other languages) which allows us to model declaration directly with the predicate calculus, without ever introducing an explicit notion of environment into the logic.

The axiomatization we obtained presents a picture of "easy axiomatizability" quite different from Hoare logic. It avoids some of the general difficulties that have always existed with Hoare logic. Since we can talk directly about the value of a variable at a given point during program execution we avoid the frequent use of auxiliary variables. For example consider the statement

$$x := 2 * x$$

To express the fact that the final value of x is twice its original value in Hoare's notation would require an auxiliary variable. In the present system it can be expressed as

$$< x := 2 * x; x > \; = 2 * x$$

As promised, the expression character of the programming language not only did not hinder the development, it was actually necessary for our notion of a Russell term to be an adequate programming logic primitive. It was essential that the programming language itself contained syntax for the value of a variable *after* the

evaluation of an expression.

Various traditionally hard problems related to declarations and aliasing disappear with the present approach. The most complex rules presented were necessary for (possibly recursive) function declarations and, to a lesser extent, loops. This should be the expected result. After all it is only these two constructs that force us to consider the issue of nontermination.

On the negative side, the main loss is the possibility of easily axiomatizing "run-time" non-deterministic and concurrent programming languages.

In many of these respects obtaining an axiomatization within this framework is similar to obtaining a denotational semantics. There are clearly other similarities as well. The notion of a Russell-term is essentially identical with that of the value component of the meaning function. Of course, differences remain. We avoid explicitly mentioning the state. (For assignment this is done by describing each component of the state. For the other constructs we could afford to be a bit less direct.) We do not explicitly need to mention environments since we can rely on predicate calculus quantification. We have replaced least fixpoint constructions by rules which directly allow us to use a form of induction in proving properties of the result. Finally we have restricted ourselves to the one form of meaning function composition afforded by the ";" connective.

The one final difference between the present logic and traditional approaches is the treatment of nonterminating constructs.

The difference is best illustrated as follows. Let α be a nonterminating expression. Assume we want to prove

$$\{\mathbf{true}\}\ \alpha;\ x := 2\ \{Q\}$$

With Hoare's partial correctness view we can prove this statement for any Q. With Dijkstra's total correctness view such a statement can never be proved. In our view the statement is valid iff it holds no matter what state α is viewed as producing. Thus we can prove the truth of the formula if Q is "x = 2".

This whole situation deserves a short explanation. Consider the task of transforming this system into a total correctness logic. This means that $\{\mathbf{true}\}\ \alpha\ \{\mathbf{true}\}$ can no longer be a valid theorem. On the other hand this is by definition equivalent to $\mathbf{true} \Rightarrow \langle\alpha\rangle\mathbf{true}$ or just $\mathbf{true} \Rightarrow \mathbf{true}$. Thus we would be led to a strange logic indeed.

The attempt to build a partial correctness logic fares no better. An argument almost identical to the one above shows that $\mathbf{true} \Rightarrow \mathbf{false}$ now has to be a theorem in our logic!

The heart of the matter is the general theorem that

$$\neg\langle a\rangle P \;=\; \langle a\rangle(\neg P)$$

In our logic this follows trivially from [<>t def]. It holds in neither the partial nor the total correctness view.

In the absence of nondeterminism this is a natural property to have. Thus if we take the view that whatever treatment of termination we chose should not interfere with nice properties which would otherwise hold, we are naturally led to an approach similar to ours[12]. From this view it is interesting to observe that it is *possible* to develop a consistent logical system in which the above theorem holds.

[12][O'Do 82] treats the issue in the same way for defined functions in Hoare logic. Yet another way of looking at our approach is as generalization of this treatment to arbitrary programming language expressions.

If we take the more utilitarian view of proving real programs within this system, our approach is still quite defensible. After all the final goal in all this is to produce a totally correct program that satisfies some given specification, usually of the form:

$$\{P\}\ a\ \{Q\}$$

We can accomplish this in several ways. We can directly express the above requirements on a within the logic as

$$T(a) \wedge (P \Rightarrow \langle a\rangle Q)$$

where T(a) is the termination predicate as defined in chapter 4. Alternatively we start by writing a program a that provably satisfies only

$$P \Rightarrow \langle a\rangle Q$$

We then see whether our proof includes termination proofs for all **while**-loops and **letrec** blocks. If so, our program is totally correct and we're done. If not we simply replace the other **while**-loops with arbitrary assignments to all state variables followed by the constant **false**. We similarly replace **letrec** blocks which are not guaranteed to terminate. In the case of these blocks we have to produce an arbitrary expression of the right signature rather than **false**. We then have a totally correct program, since the proof in effect showed that the state after execution of such a loop didn't matter. If the base logic is sufficiently nicely structured there is even some hope of doing this mechanically.

6.5. Some Problems

There are a number of language constructs which are hard to describe using our approach. We mentioned "real" nondeterminism and, in particular, concurrency. The problems related to axiomatizing unrestricted "goto" statements in Hoare logic (see [O'Do 82]) seem to be well preserved in our logic[13].

As we pointed out in chapter 2, function valued variables, and Algol 68 [Wij 75] style pointers introduce some problems we have not addressed. (There are infinitely many types. The notion of an accessible location, and thus the statement and proof of [s compactness thm] are more complicated.) Similarly the treatment of types as in [Boe 80] has been hinted at, but not fully developed. None of these problems appear insurmountable, but there is clearly work left to be done.

Certainly parts of the development we did present was not as elegant as one might have hoped.

The discussion of recursive declarations stands out. The treatment in the logic was fairly concise, but essentially required a new primitive. The denotational description is both lengthy and clumsy. In both cases the difficulty stems from the same problem: There are recursive declarations, which when viewed as equations are radically different, but which we still want to view as having the same meaning. Consider the two declarations:

letrec f == func[x] {f[x]} in ... ni

and

letrec f == func[x] {f[x]+1} in ... ni

When viewed as equations in f, the former has any function (in the right domain) as a solution, while the latter has no solutions. Nonetheless both blocks have intuitively the same meaning. Thus we cannot define the meaning of the construct as evaluating

[13]This should not be taken as convincing evidence that "goto" statements are inherently hard to axiomatize. As O'Donnell points out, there are other approaches which do not share this "problem". As expected, it is hard to deal with in systems which rely on reasoning about program fragments out of context.

the body of the block in an environment in which f is a solution to the equation[14]. In both cases we needed to resort to a fairly explicit notion of nontermination. For the axiomatization we needed a notion of "equality contingent on termination". For the denotational description we resorted to a (thinly disguised) treatment involving a special undefined value (conventionally $\perp$, here the universe). Our view that a non-terminating function is simply one that returns an arbitrary value serves us well everywhere but here.

Our treatment of the storage allocation functions NewT was also more complex than one might have hoped. This is particularly instructive since the difficulties here seemed to be largely artificial. On the one hand we can give a relatively "nice" treatment, which makes few assumptions about garbage collection etc., by using the "Is_allocated" predicate. On the other hand, our logic is sufficiently expressive to allow us to make true statements such as

<NewInt[]; NewInt[]> =
<x := 1; **while** x > 0 **do** x := x - 1; NewInt[] **od**; NewInt[]>

which cannot be proven with such an axiomatization. To make things worse, we might even argue on an intuitive basis, that there is no point to being able to prove statements like this one. The "relevant" theorems seem to be those we can obtain with the "Is_allocated" axiomatization. Thus the relative completeness theorem is forcing us into a uselessly general and complicated approach. The obvious solution would be to weaken the theorem still further. But it appears to be difficult to find a reasonable statement of such a theorem.

[14]We have not fully explored the consequences of doing so anyway. It is difficult to see how this would lead to a formal system which dealt with termination in any consistent manner.

We can compare this development with that of [Coo 76]. We use a more expressive logic. As was pointed out in the last section, this is generally desirable. This allows us to write down some formulas whose truth or falsity we would rather not specify.[15] Unfortunately the natural way of specifying which formulas we should be able to prove or disprove is by chosing not to give certain proof rules. It is much harder to give a corresponding characterization which we can use to state the completeness result. (The reader is welcome to try this for the "Is_allocated" axiomatization.) We essentially want our system to be incomplete in ways which can seemingly be characterized easily only by the logical system itself.

This leaves us in a dilemma where relative completeness begins to look like an undesirable property, but yet we would like to have some formal means of showing that we haven't arbitrarily omitted parts of an axiomatic programming language definition.

[15]Even in Cook's development such problems occur with variable allocation, but not nearly to the extent that we encounter it.

BIBLIOGRAPHY

[Bel 77] Bell; John and Moshé Machover,*A Course in Mathematical Logic*, North-Holland, 1977

[Boe 80] Boehm, H., A. Demers, and J. Donahue, "An Informal Description of Russell", Department of Computer Science, Cornell University, Technical Report TR80-430, 1980. See also [Dem 79].

[Boe 82] Boehm, Hans, "A Logic for Expressions with Side Effects", Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages, ACM, January 1982.

[Boy 79] Boyer, Robert S., and J. Strother Moore, *A Computational Logic*, Academic Press, 1979.

[Car 78] Cartwright, Robert and Derek Oppen, "The Logic of Aliasing", Department of Computer Science, Cornell University, Technical Report TR78-358, 1978.

[Cla 76] Clarke, Edmund M. Jr., "Programming Language Constructs for which it is Impossible to Obtain Good Hoare-like Axioms", Department of Computer Science, Cornell University, TR76-287, August 1976.

[Con 77] Constable, Robert L., "On the Theory of Programming Logics", Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, ACM, May 1977.

[Con 78] Constable, Robert L., and Micheal J. O'Donnell, *A Programming Logic*, Winthrop, Cambridge, 1978.

[Coo 76] Cook, Stephen A., "Soundness and Completeness of an Axiom System for Program Verification", Department of Computer Science, University of Toronto, Technical Report No. 95, June 1976.

[Cun 76] Cunningham, R.J. and M.E.J. Gilford, "A note on the semantic definition of side effects", Information Processing Letters 4, pp. 118-120, 1976.

[deB 80] de Bakker, Jaco, *Mathematical Theory of Program Correctness*, Prentice Hall, 1980

[Dem 79] Demers, Alan J., and James E. Donahue, "Data Types are Values", Department of Computer Science, Cornell University, Technical Report TR79-393.

[Dem 80] Demers, Alan J., and James E. Donahue, "Data Types, Parameters and Type Checking", Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages, ACM, January 1982.

[Dij 76] Dijkstra, Edsger W., *A Discipline of Programming*, Prentice-Hall, 1976.

[Gri 80] Gries, David and Gary Levin, "Assignment and Procedure Call Proof Rules", TOPLAS 2, Oct 1980. See also chapter 12 in

[Gri 81] Gries, David, *The Science of Programming*, Springer, 1981.

[Har 79] Harel, D., *First-Order Dynamic Logic*, Springer Verlag, 1979.

[Hoa 69] Hoare, C.A.R, "An Axiomatic Basis for Computer Programming", CACM 12, Oct. 1969, pp. 576-580.

[Jen 74] Jensen, Kathleen, and Niklaus Wirth, *Pascal User Manual and Report*, Springer, 1974.

[Kow 77] Kowaltowski, Tomasz, "Axiomatic Approach to Side Effects and General Jumps", Acta Informatica 7, pp. 357-360, 1977.

[McC 63] McCarthy, John, "A Basis for a Mathematical Theory of Computation", in *Computer Programming and Formal Systems*, ed. by P. Braffort and D. Hirschberg, North Holland, 1963. See also [Par 70].

[Mir 71] Mirkowska, G., "On Formalized Systems of Algorithmic Logic.", Bulletin de L'Academie Polonaise des Sciences, Serie des Sciences math., Astr. et phys. - Vol. XIX No. 6, 1971, pp. 421-428.

[O'Do 82] O'Donnell, Michael J., "A Critique of the Foundations of Hoare Style Programming Logics", Communications of the ACM, Vol. 25, No. 12, December 1982.

[Par 70] Park, D., "Fixpoint Induction and Proofs of Program Properties", Machine Intelligence 5, Edinburgh University Press, 1970, pp. 59-78.

[Pop 77] Popek, G. J., J. J. Horning, B.W. Lampson, J.G. Mitchell, R. L. London, "Notes on the Design of Euclid", Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, March 1977. For a description of the language see also:

Lampson, B. W., J. J. Horning, R. L. London, J. G. Mitchell, G. L. Popek, "Report on the Programming Language Euclid", SIGPLAN Notices, February 1977.

[Pri 77] Pritchard, Paul, "Program Proving - Expression Languages", Information Processing 77, North Holland, 1977, pp. 727-731. For more detail see

Pritchard, Paul A., "An Axiomatic Semantics for Expression Languages", Thesis, Australian National University, November 1979. Available as a joint technical reports from the Computer Science Departments at the Australian National University (TR-CS-80-11) and the University of Queensland (TR-20).

[Rey 81] Reynolds, John C., *The Craft of Programming*, Prentice-Hall, 1981.

[Schw 78] Schwartz, Richard L., "An Axiomatic Treatment of Asynchromous Processes in ALGOL 68", prelimin ıry draft. More detail can be found in:

Schwartz, Richard L. "An Axiomatic Semantic Definition of ALGOL 68", Computer Science Department, UCLA - 34P214-75, University of California, Los Angeles, July 1978.

[Sto 77] Stoy, Joseph E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[Wij 75] Van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., and Fisker, R.G., "Revised Report on the Algorithmic Language ALGOL 68", Acta Informatica 5, pp. 1-236.