

THESIS

2

MICHIGAN STATE UNIVERSITY LIBRARIES



3 1293 01417 2278

LIBRARY
Michigan State
University

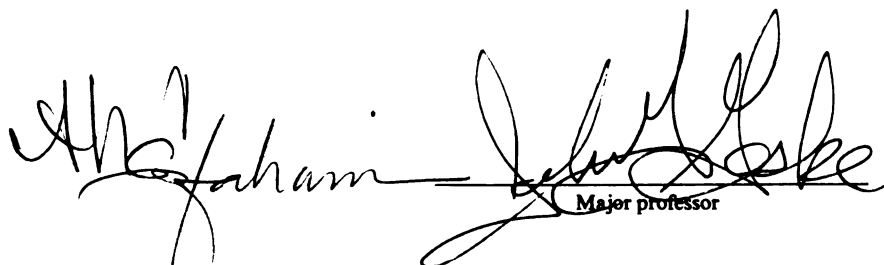
This is to certify that the
dissertation entitled
On the Formal Specification of Recursive Functions

presented by

Maureen Green Galsterer

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science


Major professor

Date April 1, 1995

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.

DATE DUE	DATE DUE	DATE DUE
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

**On the Formal Specification of Recursive
Functions**

By

Maureen Green Galsterer

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1995

ABSTRACT

On the Formal Specification of Recursive Functions

By

Maureen Green Galsterer

Current predicate transformer methods for proving program correctness deal with bounded loop programs, but these compute only a small fraction of the functions we deal with. We introduce a proof of correctness method that works for bounded, unbounded, and nonterminating loop programs. We develop a mathematical foundation that allows us to view programs as computational sequences. Using this foundation, we define iteration as a recurrence relation between predicate transformers, and we present a predicate transformer that can be used in the recurrence relation. Given this basis, we present an inductive method that can be used to prove the correctness of programs computing primitive, total, and partial recursive functions. We demonstrate the use of our method on example programs that compute functions from each class.

Copyright © by
Maureen Green Galsterer
1995

ACKNOWLEDGMENTS

It is customary in these things to acknowledge and thank the individuals who, by their commitment or love, have given more of themselves than was expected to help in this accomplishment.

First and foremost, I give thanks and all the glory to my Savior Jesus Christ, whom I serve every day as Lord.

Second, my family: To my husband John who, while he paid with money, paid with something much more valuable as well...his time; To my sons John and Jeff who supported me in a very crucial time in their lives by understanding my absences from home, who have disciplined themselves to become men that this mother is very proud of, and have kept me current with the styles so I wouldn't look like a geek; To my parents Robert and Irene Green...I've kept my family name all these years just to honor you.

Third, to my advisor, Dr. John G. Geske. You are a gentleman and a scholar. Your direction and guidance was crucial. But even more important (and something only other women in academia will understand) you accepted me as a gentlewoman and a scholar. I cannot pay you high enough tribute, so I'll just say: Thank you, and I hope we'll be friends forever. I also want to thank my committee members, Dr. Betty H.C. Cheng and Dr. Abdol H. Esfahanian, for their support.

Fourth to my pastors: Rev. Judith K. Shepherd, and Rev. Jud Silveus. Thank you for the times you've prayed with me to see this through, and even more for all the times you've sought God on my behalf that I don't even know about. I promise you I'll use this degree to build the Kingdom.

Fifth to my sisters and brothers in Christ at MSU: Barb Birchler, Barb Czerny, Dr. Jon Englesma, Sally Howden, Ron Sass, and Dr. Steve Walsh...You've all been bright points of encouragement and comfort. To you who have graduated, God Speed! To you who will follow me, Take Heart! If we can do all things through Christ, you can certainly trust Him to complete your research.

Finally, to my very best friend in Computer Science, Dr. Steve Turner...I love you, Steve...ack!Phphhpfttt!

TABLE OF CONTENTS

LIST OF FIGURES	vii
1 Introduction	1
1.1 Motivation for the Research	2
1.1.1 Statement of the Problem	3
1.2 Overview of the Solution	4
1.3 Organization of the Thesis	8
2 Mathematical Preliminaries and Definitions	9
2.1 Relations	9
2.1.1 Definitions	9
2.1.2 Operations	10
2.2 Properties of Relations	11
2.2.1 Ordering Properties	11
2.2.2 Lattices	12
2.2.3 Measuring the Information Contained in a Relation	12
2.3 Functions	13
2.4 Assertions as Approximations	14
3 Previous Work	15
4 Semantics of A Language	26
4.1 Syntax of the Language	27
4.2 Semantics of the Language	29
4.2.1 The Denotational Semantics of the Language	29
4.2.2 The Operational Semantics of the Language	35
4.2.3 Semantic Equivalence	44
5 Mathematical Foundation for Reasoning About Nonterminating Computations	48
5.1 The Semi-Lattice of Computations	52

5.2	Fixed Points	55
5.2.1	Interpretation of Fixed Points	56
5.3	Theory of Fixed Points	61
5.3.1	Monotonic Functions	61
5.3.2	Properties of Monotonic Functions	62
5.3.3	Least Upper Bound	64
5.3.4	“;” When Viewed As A Functional Is Continuous	66
5.3.5	Fixed Points of the Functional τ	72
6	The <i>wpw</i> Predicate Transformer	74
6.1	Macros	75
6.1.1	Syntax of Macros	77
6.1.2	Use of Macro Names	78
6.2	The <i>wpw</i> Predicate Transformer	79
6.2.1	Formal Definition of an Unbounded Loop	81
6.2.2	Incremental Progress	82
6.2.3	Properties of the <i>wpw</i> Predicate Transformer	84
6.2.4	Relationship Between I , R' , and R	87
7	Application of the <i>wpw</i> Predicate Transformer	89
7.1	Description of the Method	89
7.2	Examples	91
7.2.1	Proof of Correctness of a Total Recursive Function	91
7.2.2	Proof of Correctness of a Partial Recursive Function	98
7.2.3	Proof of Correctness of a Nonterminating Computation	102
7.3	Annotated Program Code for Examples	108
8	Conclusion	113
8.1	Future Work	114
	BIBLIOGRAPHY	117

LIST OF FIGURES

4.1	Syntax of the Language.	28
4.2	Example of the denotational definition of <code>whiledo od</code>	35
4.3	Random Access Machine (RAM).	37
7.1	Example 1, A Total Recursive Function.	108
7.2	Major Macro for Example 1.	109
7.3	Minor Macro for Example 1.	110
7.4	Example 3, A Nonterminating Program.	110
7.5	Example 3, The Macro <i>divides</i>	111
7.6	Example 3, The Macro <i>Prime</i>	112

CHAPTER 1

Introduction

This research deals with partial recursive functions. Specifically, we are interested in reasoning about programs that use unbounded iteration or general recursion to compute partial functions. A large body of research already exists on reasoning about terminating programs that stems from Dijkstra's *weakest precondition* (*wp*) predicate transformer [1]. All of this work deals with primitive recursive functions since termination must be guaranteed with an *a priori* upper bound. This restriction was based not only on Dijkstra's firm belief that all meaningful programs terminate [2], but for simplicity, because it allowed him to define iteration as a recurrence relation between predicates [1].

We will show that a mathematical foundation for reasoning about unbounded and nonterminating programs exists. Based on this foundation, we will define iteration as a recurrence relation between predicate transformers. We will define a new predicate transformer that can be used with our mathematical foundation to prove the correctness of programs computing primitive recursive, total recursive, and partial recursive functions.

1.1 Motivation for the Research

While many programs have *a priori* time bounds, there are important programs that do not, such as those that manipulate sequential files. These programs are expected to terminate, but they require unbounded loops. It is also common to use programs that are not designed to terminate at all, such as operating system daemons and networking processes. *A priori* time bounds are meaningless for programs using unbounded loops to compute total functions, and for programs using nonterminating loops to compute partial functions.

Program specification techniques based on predicate logic use assertions to precisely define the condition of a program's variables before and after a program statement executes. These assertions are called *preconditions* and *postconditions*, respectively. The *weakest precondition* (*wp*) calculus of Dijkstra is one of the most widely known of these methods that use predicate transformers. Dijkstra's predicate transformer maps a program statement and its postcondition to a predicate describing the set of all initial states from which the statement can execute leaving its postcondition true. Dijkstra envisioned his method as a top-down design tool. The *wp* calculus is a total correctness method, *i.e.*, it is only applied to statements that are guaranteed to terminate. This guarantee ensures that the postcondition of any statement can be reached and verified. This assurance of always being able to reach a postcondition allowed Dijkstra to define the semantics of simple iteration as a recurrence relation between predicates [1], keeping it in the realm of first order logic, but it also restricts the utility of the method: The only recursions that can be proven to terminate *a priori* are direct (tail) recursions. To encompass all the computable functions we need general recursion.

In the classes characterized by general recursion it is not possible to predict when, or if, a computation will terminate. Any method for reasoning about correctness in

these classes cannot make termination an issue in its proofs, therefore our solution does not depend on termination. We can deal with termination if it does occur, but we concentrate on correctness at each step of a computation not just at its end. So, whether a computation step leads to a final state or not we can verify its correctness.

Proof of correct design was Hoare's intention when he proposed predicate logic with equality as a means for reasoning about programs [3]. Hoare viewed a program as a logical model and used predicate logic to show that as a sequence of statements it was a tautology. Dijkstra's main disagreement with Hoare was that programs are not recipes for computing, but are agents that when run on a computer can effect a computation, *i.e.*, programs execute [2]. Dijkstra contended that proving a computing method is correct is only half of a proof of correctness. In addition, he insisted that showing termination with correct results was necessary.

1.1.1 Statement of the Problem

Proof of correctness methods that rely on the *wp* calculus are only suited to programs computing functions in the primitive recursive class because the *wp* predicate transformer is too severely structured to accomodate the more powerful classes of functions. The *wp* predicate transformer takes a program statement and its postcondition as its arguments. Termination must be guaranteed *a priori* to ensure that the postcondition can be reached and verified, and the primitive recursive functions are the only class of functions for which *a priori* time bounds exist. *A priori* time bounds are meaningless in the classes characterized by general recursion, but limiting proof of correctness methods to the small class where the *wp* calculus works limits our ability to verify the correctness of programs that we use regularly, *i.e.*, all programs that use general recursion or unbounded looping to compute.

We will show that a mathematical foundation for reasoning about unbounded and nonterminating programs exists. Based on this foundation, we will define iteration as

a recurrence relation between predicate transformers. We will define a new predicate transformer that can be used with our mathematical foundation to prove the correctness of programs computing primitive recursive, total recursive, and partial recursive functions.

1.2 Overview of the Solution

Our research carries Dijkstra's statement that "programs execute" to its logical conclusion. Dijkstra ignored actual execution in favor of its beginning and its end, concentrating on the predicate logic assertions he could make about the initial and final states of a computation. We use the results of each stage of a program's execution and verify correctness from program state to program state; we use the entire sequence of a computation. We will show that there is a logical foundation for reasoning about unbounded and nonterminating computations. We will provide the mathematical foundation for reasoning about such computations, and we will give a concrete method that can be used in proofs of correctness for programs using loops with and without *a priori* upper bounds. Our method is based on a definition of iteration as a recurrence relation between predicate transformers, *i.e.*, functions from predicate transformers to predicate transformers. Using our predicate transformer, we will show with examples that we can reason about nonterminating and terminating programs.

Using computational sequences raises technical issues because a consistent foundation must be carefully laid, and it raises theoretical issues because proofs of correctness are based on second order logic. We quantify over predicates, so we define iteration as a recurrence relation between predicate transformers. While this adds complexity to our foundation because we must reason in second order logic, it extends our ability to prove correctness of total and partial recursive functions, as well as primitive

recursive functions.

Technically, when we deal with a computational sequence we must first define the environment that generates the sequence. Because the sequence represents a “run” of the program we must have a machine to execute it and a language to express it. The machine we define must be sufficiently general so as to represent the class of machines available in the real world, so we use the Random Access Machine of Meyer and Ritchie [4]. The language we need must be simple but must also form a complete programming system with the machine. Our language is similar to Dijkstra’s except that we replace his bounded *DO* loop with an unbounded *WHILEDO*.

The semantics of our language are critical because we must be able to express the meaning of terminating and nonterminating computational sequences with defined results (sequences that represent the computations of total and defined partial functions, respectively). We must also be able to express the meaning of nonterminating computational sequences with undefined results (sequences that represent partial functions undefined on their input). We provide a formal denotational semantics for our language that admits an undefined element to handle partial functions in a fashion similar to de Bakker [5] and Nelson [6]. The operational semantics of our language is the foundation for our computational sequences. The sequences must be sufficiently abstract so as not to make the method dependent on any particular implementation but they must make sense given the environment we have defined. We must also guarantee that the information represented by the variable valuations in each element of the sequence logically succeeds that of the element before it, that it represents a useful value (*i.e.*, it was derived as a result of a valid computation step and not as a side effect), and that it represents a true approximation to the limit of the sequence itself.

To ensure all this we differentiate operationally between the function computed by a program and the operational meaning of a program. The *function computed*

by a program is the value of the variables in a final state or an undefined value if the program does not terminate. This parallels the usage of the valuation function in the denotational semantics definition. The *operational meaning of a program*, or what it means to compute on our machine, is the computational sequence returned by our operational semantic function. Our function allows us to express formally the notion that computers actually execute programs. Making this distinction allows us to prove that the operational semantics and denotational semantics are equivalent, and to show that a computation produces more information than can be expressed by its initial state/final state pair.

Defining the meaning of a program to be a computational sequence gives us a series of program states to examine. The equivalence of our denotational and operational semantics ensures that our programs will not display side effects. This is not sufficient to guarantee that each state of a computation logically succeeds that of the element before it or that each successive state is a better approximation to the limit of the computation than its predecessor. For these last two requirements we must impose a well-ordered structure on our sequences; we must guarantee that in all cases our sequences are chains. We do this by organizing our state space as a complete lattice, and then proving that our program statements are monotonic (this ensures that each state is a valid extension of its predecessor so no unanticipated results will creep into our sequences) and that both our program statements and the higher order function used by our operational semantics to build our sequences are both monotonic and continuous (this ensures that each element of a chain is a true approximation to the limit of the computation).

Kleene's First Recursion Theorem holds that every computational chain has a least upper bound [7]. We can use this fact to induct over the elements of our computational chains in our proofs of correctness. We can step from predicate transformer to predicate transformer to verify that each element of the chain is an approximation

to the least upper bound. For terminating computations the least upper bound is the final state. It is defined by a logical assertion that fulfills the program's specification when the computation is correct. For nonterminating computations we use the least upper bound of each subsequence in the chain, and view the logical descriptions of these elements as approximations to the predicate that defines the limit of the computation itself. To do this we have recognized that for every looping construct there exists a logical assertion that is an approximation to a defined loop's specification. If termination occurs the conjunction of the loop invariant, this approximation, and termination implies the postcondition. If termination has not occurred, the truth of the conjunction of the loop invariant and this approximation proves the correctness of the current iteration of the loop.

The theoretical issue that arises from the use of computational chains to prove correctness is that the predicate transformer we define provides a mapping between the logical assertions that define each element of the chain, not just for the initial and final states as the *wp* predicate transformer does. For noniterative commands the operator behaves like the *wp* predicate transformer we are accustomed to because such commands are atomic (they have an initial state and a final state only). For iterative commands the operator defines a recurrence relation between predicate transformers because it maps the predicate transformer of one state to the predicate transformer of another. We will show in Chapter 6, Section 6.2.3 that the predicate transformer we define has all of the properties that the *wp* predicate transformer has.

We show that, given the syntax of unbounded looping and semantics to express the results of partial functions, a logical basis for reasoning about general recursive computations exists, and can be extended into a method for proving that they are correct.

1.3 Organization of the Thesis

In Chapter 2 we present mathematical definitions and preliminaries that we use. In Chapter 3 we review the related work that led to the development of the *wp* calculus. In Chapter 4 we define our machine model and its language, give our denotational and operational semantics, define formally the distinction we make between the operational value of a program and the operational meaning of a program, and prove that our denotational and operational semantics are equivalent. In Chapter 5 we present the complete lattice induced by our operational semantics and prove the properties of monotonicity and continuity for our program statements and the higher order functional we use to construct computational chains. In Chapter 6 we introduce our predicate transformer with the Incremental Progress Theorem, and give its properties. We show how we can use this theorem in our inductive verification method. In Chapter 7 we provide examples of the method used on a total recursive function, a partial recursive function we expect to terminate, and a partial recursive function that is nonterminating by design. In Chapter 8 we discuss several areas of application for this research as future work.

CHAPTER 2

Mathematical Preliminaries and Definitions

2.1 Relations

2.1.1 Definitions

Like Dijkstra [1], Hehner [8], and Nelson [6] we view commands as binary relations on the set of program states. This state space is composed of machine vectors of length n , where n is the total number of variables used by a program. Thus a program command maps one program state (valuation of the machine vector before the command executes) to another state (valuation of the machine vector after the command executes).

A command, C , on the set Γ of program states is a subset of $\Gamma \times \Gamma$. When we represent individual elements of Γ we write γ to mean $\gamma(< x_1, x_2, \dots, x_n >)$, the valuation of the machine vector. Let $(\gamma, \gamma') \in C$; γ is an *argument* in C and γ' is an *image* in C . The *domain* of relation C is the set of arguments of C ; formally, $dom(C) = \{\gamma | \exists \gamma' : (\gamma, \gamma') \in C\}$. The *range* of relation C is the set of images of C ; formally, $rng(C) = \{\gamma' | \exists \gamma : (\gamma, \gamma') \in C\}$.

The *identity* relation $id_\Gamma = \{(\gamma, \gamma') | \gamma \in \Gamma \text{ and } \gamma' = \gamma\}$, and the *empty* relation $\emptyset = \{\}$ are constant relations.

A *relational* is a relation on the set of binary relations (i.e., commands) on Γ .

2.1.2 Operations

Since commands are binary relations, we can obtain other relations using *union*, *intersection*, and *difference* on these binary relations. In addition to these operations, we define the following operations on relations.

- **Inverse:** The *inverse* of command C is the relation denoted by C^{-1} and equal to $\{(\gamma, \gamma') | (\gamma', \gamma) \in C\}$.
- **Composition:** Let C_1 and C_2 be two commands on Γ . The *composition* of C_1 with C_2 is the relation denoted by C_1C_2 and defined by

$$C_1C_2 = \{(\gamma, \gamma') | \exists \gamma'' : (\gamma, \gamma'') \in C_1 \wedge (\gamma'', \gamma') \in C_2\}$$

- **Iterated Composition:** Let C be a command on Γ . The i^{th} *iterated composition* of C , for $i \geq 0$, is the relation denoted by C^i and defined by

$$C^0 = id_\Gamma,$$

$$C^i = C^{i-1}C, \text{ for } i \geq 1.$$

Note that for all $i \geq 1$, $dom(C^i) \subseteq dom(C)$.

- **Transitive closure:** Let C be a relation on Γ . The *transitive closure* of C is the relation denoted by C^+ , and defined by

$$C^+ = \{(\gamma, \gamma') | \exists i \geq 1 : (\gamma, \gamma') \in C^i\}.$$

- **Reflexive transitive closure:** The *reflexive transitive closure* of command C on Γ is the relation denoted by C^* and is equal to $id_\Gamma \cup C^+$.
- **Image set:** The *image set* of $\gamma \in \Gamma$ of command C is the set denoted by $\gamma.C$ and defined by

$$\gamma.C = \{\gamma' | (\gamma, \gamma') \in C\}.$$

A command's specification defines its image set.

To extend this notation, if $\Gamma' \subseteq \Gamma$ then $\Gamma'.C = \bigcup_{\gamma \in \Gamma'} \gamma.C$. We define $C.\gamma$ and $C.\Gamma'$ as $\gamma.C^{-1}$ and $\Gamma'.C^{-1}$.

- **Nucleus:** The *nucleus* of command C is the relation

$$N(C) = CC^{-1}.$$

This relation contains the state pair (γ, γ') if and only if γ and γ' map to the same state γ'' under C .

The set of elements that make up the nucleus can be defined by the **weakest precondition** of a command and its specification.

2.2 Properties of Relations

2.2.1 Ordering Properties

Let C be a relation on Γ . C is said to be *reflexive* if and only if $id_\Gamma \subseteq C$. C is said to be *transitive* if and only if $C^2 \subseteq C$. C is said to be *antisymmetric* if and only if $C \cap C^{-1} \subseteq id_\Gamma$.

C is said to be a *partial ordering* if and only if it is reflexive, antisymmetric, and transitive.

2.2.2 Lattices

Let C be a partial ordering relation on Γ , and let Γ' be a subset of Γ . An element γ of Γ' is said to be C – *maximal* in Γ' if and only if there is no γ' other than γ in Γ' such that $(\gamma, \gamma') \in C$.

Let C be a partial ordering relation on Γ , and let Γ' be a subset of Γ . An element $\gamma \in \Gamma'$ is said to be C – *minimal* if and only if there is no γ' other than γ in Γ' such that $(\gamma', \gamma) \in C$.

Let γ and γ' be two elements of Γ and let C be a partial ordering on Γ . The set of *upper-bounds* of γ and γ' is denoted by $ub(\gamma, \gamma')$ and defined as

$$ub(\gamma, \gamma') = (\gamma.C) \cap (\gamma'.C).$$

The set of *lower-bounds* of γ and γ' is denoted by $lb(\gamma, \gamma')$ and defined as

$$lb(\gamma, \gamma') = (C.\gamma) \cap (C.\gamma').$$

Let γ and γ' be two elements of Γ , and let C be a partial ordering relation on Γ . A C – *least upper bound* of γ and γ' , if it exists, is a C -minimal element of $ub(\gamma, \gamma')$, denoted $lub(\gamma, \gamma')$. A C – *greatest lower bound* of γ and γ' , if it exists, is a C -maximal element of $lb(\gamma, \gamma')$, denoted $glb(\gamma, \gamma')$.

Let C be a partial ordering on Γ . We say that C is a *lattice* if and only if every pair $(\gamma, \gamma') \in \Gamma$ has a greatest lower bound and a least upper bound.

2.2.3 Measuring the Information Contained in a Relation

Let C and C' be two relations on Γ . We say that C is *more-defined* than C' (or C' is *less defined* than C) if and only if

1. $dom(C') \subseteq dom(C)$,

$$2. \forall \gamma \in \text{dom}(C'), \gamma.C \subseteq \gamma.C'$$

Example 2.1 *Suppose there are two observers watching a deterministic input/output process. The process draws inputs from the set $\{a, b, c, d\}$ and has outputs in the set $\{0, 1, 2, 3, 4\}$. Suppose C and C' are the reports of the two observers as they view the process. Let*

$$C = \{(a, 0), (b, 1), (c, 2), (d, 3)\}$$

$$C' = \{(a, 0), (a, 1), (a, 2), (b, 1), (b, 2), (b, 3), (c, 2), (c, 3), (c, 4)\}$$

Report C is more-defined than report C' because the observer who gives it reports about more inputs than the observer who gives report C' . This is what condition 1 means. Report C is more-defined than report C' for the additional reason that the observer who gives it is more precise in his assignment of the inputs both have recorded. This is what condition 2 means.

2.3 Functions

Functions can be viewed as relations that map elements of the domain to unique elements of the range. For a function f and domain element x , $x.f$ is the singleton set denoted $f(x)$.

We use a function name f in three distinct ways. First, f denotes a set of pairs such that for any value x there is at most one pair (x, y) . Second, $x f y$ holds if (x, y) is in f . Third, $f(x)$ is the value associated with x , that is, $(x, f(x))$ is a member of the function (relation) f . Notice that the usage $x f y$ denotes a predicate.

We define functions as restricted forms of relations because then the theory and terminology for relations carries over to functions. Thus we know what image set and iterated composition of a function mean. Since a function is a relation we know what the inverse f^{-1} of function f is. f^{-1} is a function when f is invertible, i.e., when f is one-to-one and onto.

Nucleus. The nucleus of a function f is

$$N(f) = ff^{-1};$$

which can also be written

$$N(f) = \{(x, x') | f(x) = f(x')\}.$$

2.4 Assertions as Approximations

Each state in a computational chain represents an approximation to the limit of the computation. These states can be described by logical assertions, and each logical assertion is approximated by its predecessors. We represent predicates with capital letters of the roman alphabet: $A, B, \dots, P, Q, R, \dots, Z$. When we represent the approximation to a particular predicate, we append a prime ($'$) to the letter we use for that predicate. For example, R' approximates R .

This convention is used when we specify a looping construct. For example:

$\{Precondition\}$

$\{Invariant\}$

while \mathcal{G} **do**

S ;

$\{R' : \textit{Assertion about } S\}$ this assertion approximates R

od

$\{R : \textit{Assertion about whiledo} \dots \textit{od}\}$

CHAPTER 3

Previous Work

The application of mathematical techniques to prove the correctness of a program was proposed in 1969 by C. A. R. Hoare [3]. Hoare's intention was to provide a logical basis for proofs of the properties of a program. His primary concern was to demonstrate that a given program is able to carry out its intended function. Hoare considered the semantics of computation without addressing the implementation issues. Since Hoare was interested in the axiomatic semantics his interest was primarily model theoretic. Hoare made a distinction between a program and the implementation of the program, sidestepping the issue of termination of a computation. Hoare's work defined the notion of *partial correctness*, that is, programs should guarantee that they will not produce an incorrect answer. This gave the first half of the requirements used today to prove that programs are correct.

Hoare specified the intended function of a program by making general assertions about the values which the relevant variables will take after execution. The assertions are not specific about particular values each variable will have, but rather specify certain general properties of the values and the relationships that hold between them. Hoare's notation is predicate logic.

Hoare recognized that the validity of the results of a program will often depend on the values taken by the variables before the program is initiated. He used statements

in predicate logic to specify the “initial preconditions” of successful use and also to describe the results obtained on termination. To state the required connection between a precondition (P), a program (Q), and a description of the result of its execution (R), Hoare introduced the notation

$$P\{Q\}R,$$

called a “Hoare triple.” The interpretation of this statement is: If the assertion P is true before initiation of the program Q , then the assertion R will be true on its completion.

Hoare defines one axiom schema to specify the assignment statement, and three rules: one to specify the decision construct, one for composition of sequential statements, and one to cover iteration.

Building on Hoare’s work in the mid 70’s, Dijkstra [2], [1] observed that computer programs were not just static sequences of statements, but when programmed on a computer became dynamic entities. Where Hoare looked at programs as sets of statements to be verified with predicate logic, Dijkstra was concerned with the outcome of statements after their execution. Dijkstra insisted that the only programs we are interested in are the ones that terminate and accomplish what we want [2]. Dijkstra’s work defined the notion of *total correctness*: partial correctness (provided by Hoare) and termination (insisted upon by Dijkstra).

Dijkstra’s *weakest-precondition calculus* is probably one of the most familiar of the specification methods that use predicate transformers to deal with program statements and control structures. Whereas Hoare was interested in correctness of the method, Dijkstra was interested in the correctness of the method *and* correctness in the implementation of the method. Dijkstra recognized the utility of Hoare’s work in specifying the preconditions and results of executing a statement (which he

called “postconditions”), so his method used predicate logic to specify the pre- and post-conditions of the four basic programming constructs: assignment, sequencing, decision, and repetition.

Unlike Hoare, Dijkstra was very interested in what actually happens during execution of one of these basic programming statements, so he introduced notation that could capture the meaning of “implementing a statement.” Dijkstra defines the semantics of a programming construct in terms of *weakest preconditions* (*wp*). In this notation, $\{Q\}S\{R\}$ means that execution of statement (program) S begun in any state satisfying predicate Q will terminate in a state satisfying predicate R . The different syntax used to present this assertion (or “Dijkstra triple”) is not accidental: Dijkstra strengthened the meaning of the assertion from correctness (Hoare’s meaning) to correctness and termination. In this context, Q is called the *precondition* and R is called the *postcondition* of a statement S . The *weakest precondition* of S with respect to R , $wp(S, R)$, represents the set of all states such that execution begun in any one of them will terminate with R true. The notation $\{Q\}S\{R\}$ is another notation for

$$Q \Rightarrow wp(S, R),$$

which is a statement in predicate calculus that is either true or false in any state.

The syntax of the four basic programming statements is as follows. An assignment statement is denoted by “ $x := E$ ” where x is any variable and E is any expression of the appropriate type. Sequencing is denoted by “ $S1; S2$,” first execute $S1$ and then execute $S2$. The decision construct, *IF* is denoted by

$$\text{if } B_1 \rightarrow SL_1 \square \dots \square B_n \rightarrow SL_n \text{ fi}$$

where the B_i are Boolean expressions called “guards,” and the SL_i are statements (or statement lists) called “alternatives.” To execute an *IF* statement, execute any

one of the alternatives whose guard is true.

The repetitive construct, *DO*, denoted

$$\text{do } B_1 \rightarrow SL_1 \square \dots \square B_n \rightarrow SL_n \text{od}$$

is executed as follows: repeatedly, as long as some guard B_i is true, execute any one of the alternatives whose guard is true. The repetitive construct only terminates in a state in which none of the guards is true. When the repetitive construct has terminated properly, we know that all of its guards are false.

This definition seems appropriate for unbounded loops but it lacks the mechanism for proving termination that Dijkstra required. Dijkstra supplied this mechanism by building the requirement of termination into his semantic definition of the above syntax. The semantic meaning of the construct is: For at most k iterations, execute any one of the guards that is true. Since the repetitive construct can only terminate when none of the guards is true, the *a priori* upper bound k is used to design the guards so that this condition is ensured.

Dijkstra used predicates to define sets of initial and final states the way Hoare did [3]. The main difference in their methods is that while Hoare introduced sufficient preconditions such that the mechanisms will not produce the wrong result (but may fail to terminate), Dijkstra introduced necessary and sufficient, *i.e.*, “weakest” preconditions, such that the mechanisms are guaranteed to produce the right result. The semantic tool he used was predicate transformers, which specify, for a given statement S (which he called a “mechanism”) and postcondition R , the weakest precondition guaranteeing that S will establish R .

The weakest precondition (semantic) equations for the four basic programming constructs are:

1. Assignment: $wp(“x := e”, R) = R_e^x$.

Where R_e^x indicates that e is defined on the domain of R and we replace every occurrence of x in R with e .

2. Sequential composition: $wp("S1; S2", R) = wp("S1", wp("S2", R))$.

3. IF : $wp(IF, R) = (BB \text{ and } (\forall i : 1 \leq i \leq n : B_i \Rightarrow wp(SL_i, R)))$.

Where BB is a Boolean expression that ensures there exists a true guard, and the second term requires that each guarded list eligible for execution will lead to an acceptable final state.

4. DO : $wp(DO, R) = (\exists k : k \geq 0 : H_k(R))$.

Where $H_k(R)$ is the weakest precondition guaranteeing proper termination after at most k selections of a guarded list, leaving the system in a final state satisfying R . $H_k(R)$ is defined by cases:

- $H_0(R) = R \wedge \neg(\exists j : 1 \leq j \leq n : B_j)$
- $H_k(R) = wp(IF, H_{k-1}(R)) \vee H_0(R)$

The semantics of Dijkstra's looping construct are defined using tail recursion, with the predicates describing the state of the computation at each level of recursion. Dijkstra had to show that his recursions formed a finite decreasing chain to ensure termination of his looping constructs. This is the basis for the definition of *computational progress* which is measured as a decrease in an integer-valued function that is bounded from below. Computational progress is perfectly suited for functions in the primitive recursive class because the primitive recursive functions are defined to be those functions that are computable by *bounded loop programs* [9]. Dijkstra introduced his finite integer bound function as a free variable that must be manipulated in each proof of a repetitive construct.

Hehner [8] challenged the utility of the repetitive DO statement and offered the notion of *recursive refinement* for specifying looping statements. He argued that the

semantics of Dijkstra's *DO* were the most complicated part of Dijkstra's language, because of the requirement of demonstrating a finite decreasing chain of predicates by manipulating another variable in the semantic equations. Hehner reasoned that since Dijkstra was defining the semantics of the looping construct with recursion, it would be easier to throw away the looping construct entirely and just use recursion. He proposed replacing Dijkstra's looping construct with recursive refinement, a technique that uses divide-and-conquer to repeatedly define a repetitive problem in finer and finer detail until the base definition of the problem is reached. At this point, the specification is composed entirely of sequential steps which can each be specified individually without regard to termination, since it is assumed that each sequential action terminates. In this sense, refinement eliminates loops.

In eliminating the looping construct, Hehner eliminated the need to carry along and manipulate an explicit counter in his semantic equations. This removed the added complexity of demonstrating that the recursive computation behaved as a finite decreasing chain. It is important to note that Hehner explicitly specified tail recursion, which was appropriate because his motivation was to replace Dijkstra's bounded loops. Limiting the method to tail recursion had two consequences: first, the method could only be used to specify primitive recursive functions; second, the method had to be designed to demonstrate termination. Hehner addressed the second consequence with the same mechanism Dijkstra used: computational progress, incorporating the requirement to show computational progress into the refinement process.

The recursive refinement technique involves the invention of a name for a portion of a program, using the name in place of the program portion, and specifying the text of the program portion elsewhere. Hehner referred to the use of a name in place of some statements as a "call," and referred to the specification of the statements as a "refinement." A call is a name enclosed in quotation marks; a refinement consists of the quoted name, followed by a colon, followed by a statement list (SL):

"name":SL

Hehner's "call" gives no semantic equation; a call is given meaning by the details of its refinement. A refinement gives the equation: $wp(\text{"name"}, R) = wp(SL, R)$.

Computational progress is the cornerstone of Dijkstra's repetitive construct and Hehner's recursive refinement because both methods rely on it in their proofs of termination. In its most formal sense, computational progress refers to a theorem of Dijkstra's that shows how a loop invariant and a finite integer bound function can be used to prove that a repetitive construct is totally correct, i.e., is partially correct and terminates.

Informally, computational progress refers to the manipulation of a loop counter to demonstrate that a repetitive construct will make progress towards termination.

In all of its manifestations, computational progress is an explicit reference to an *a priori* upper bound on the number of iterations a loop must perform. Because of this, every method that depends on computational progress is limited to programs computing in the primitive recursive class of functions.

The primitive recursive class contains only a fraction of the functions we rely on regularly. Consider this Pascal program segment

```
while not eof(datafile) do
  readln(datafile, value);
```

This loop depends on the file variable reaching a particular condition independent of a counter, and cannot be proven to meet its specification using the *wp* calculus or recursive refinement. There are many such applications outside the primitive recursive class: communicating processes, searching, operating systems processes, networking services, etc.

de Bakker addressed the problem of unbounded loops in 1980 [5]. He added an unbounded looping construct to Dijkstra's language and defined a denotational semantics to formally express the function computed by each statement. To accomodate

nonterminating loops he extended the state space of program variables to include an undefined element (\perp), and called it “bottom” in keeping with Scott’s work on denotational semantics [10]. Thus his program statements as functions could return either a defined value from the program’s state space of all variable valuations, or they could return the value \perp . de Bakker defined an operational semantics that acted as a function, returning either a state whose variables had the valuation resulting from correct execution of a statement, or in the case of a nonterminating statement, a state with variables undefined.

The availability of both an initial and a final state for any execution of a statement fulfilled the requirements of Dijkstra’s *wp* predicate transformer. However, de Bakker did not redefine the foundation of Dijkstra’s predicate transformer, so the predicate logic meaning *FALSE* that defined \perp violated one of the properties of Dijkstra’s predicate transformers: $wp(\text{Statement}, \text{FALSE}) = \text{FALSE}$, i.e., statements cannot lead to undefined states.

Given the proper motivation this property of the *wp* predicate transformer can be redefined. In fact this was done subsequently by Nelson [6]. A more serious concern that arose from de Bakker’s adherence to the initial state/final state pair required by the *wp* predicate transformer is that loops designed to be nonterminating, such as operating system processes, require the postcondition *FALSE* by definition of his denotational semantics, and his operational semantics was designed to return only an undefined state as well. The problem with this is that intuitively much more can be said about nonterminating computations that are defined on their input, but all de Bakker could say, in the absence of intermediate states to analyze, was that the loop did not terminate.

In 1984 Hehner worked briefly at defining program statements themselves as predicates in an attempt to maintain the definition of iteration as a first order recurrence (a recurrence between predicates) [11]. Motivated by this, Nelson addressed the prob-

lem of recursively computed partial functions by generalizing the *wp* calculus [6]. In Dijkstra's world all loops terminate so they have descriptive predicate logic (i.e., not *FALSE*) postconditions. This is the essence of his *Law of the Excluded Miracle* ($wp(\text{Statement}, \text{FALSE}) = \text{FALSE}$). Nelson rejected this "law," claiming that since nonterminating recursions never reach a final state they can be specified by the predicate logic description *FALSE*, denoting the empty set. With the predicate logic description *FALSE* for postcondition of a nonterminating computation Nelson had both the set of initial states and the set of final states (although it was empty) he needed to satisfy the *wp* predicate transformer. However, the postcondition *FALSE* is not helpful for verifying correctness, unless the computation is for a partial function undefined on its input, and since it is not even possible to identify these computations, Nelson could certainly not predict them in advance.

Therefore, Nelson focused on the partial correctness conjunct of the *wp* predicate transformer. This conjunct, the *weakest liberal precondition (wlp)* expresses that a computation will not produce an incorrect answer, i.e., each stage is not wrong. This is not equivalent to saying that each stage is "right," because this stronger statement requires termination of the whole in the *wp* model. Nelson defined a recursive computation as a sequence of approximations to the limit and used a pair of predicates to specify each element of the sequence, making the mapping of the predicate transformer a recursion between predicates. One predicate expressed what the variables were not, and the other assured that the recursion would halt. He showed that his work could be applied to direct recursion and implied that it could be tailored to general recursion, although the assurance of halting only works if you have a hand on the power switch in this case.

Nelson provided the mathematical foundation for extending the partial correctness predicate transformer (*wlp*) into the classes represented by general recursion. Nelson's work relied on a limit theorem that required only monotonicity (not mono-

tonicity and continuity) because his computational model was based on recursion. A general recursive computation can be modeled as a tree because the computation is allowed to backtrack, or recover, from “dead ends.” Continuity limits the number of dead ends that can be recovered from; it precludes unbounded nondeterminism. Nelson showed that as a computation progresses correctly monotonicity ensures that no erroneous values are admitted into the set of states that represent the computation. He showed that the *wlp* predicate transformer is an accurate map from the states in the computation sequence to the initial states that allow them.

Because the *wp* predicate transformer only discusses initial and final states Nelson’s generalization of it could not produce a method for verifying the sequence at any intermediate stage. Additionally, his lack of continuity, coupled with the fact that his predicates expressed the compliment of the actual computational sequence precluded him from making assertions about individual states of the computation itself. The *wlp* predicate transformer only guarantees what a computation is not doing (i.e., behaving incorrectly). We cannot use it of itself to say anything about what the values in any stage actually are, because the *wlp* predicate transformer guarantees correctness only upon termination.

Hesselink also worked with recursive computations [12]. He developed a language that included procedure declarations that supported mutual recursion and gave its semantics in terms of the *wp* calculus. He used the syntax of his procedure declarations to reason in his semantic equations, but not in his program specifications as we do. Reliance on the *wp* predicate transformer only allowed him to show partial correctness in nonterminating computations. However, this is still more than Dijkstra could do. Like Nelson, Hesselink showed that the *wlp* predicate transformer is valid throughout a general recursive computation.

His result was stronger than Nelson’s because the version of the Knaster-Tarski limit theorem he used allowed him to show that the Law of the Excluded Miracle

can be preserved while unlimited backtracking is supported. This work agreed with Dijkstra and van Gasteren [13] that the restriction to continuity can be lifted in the *wp* model. However, use of the *wp* model with its reliance on termination limited Hesselink to the initial state/final state pair, so like Nelson, all he could say about a nonterminating computation was that it would not be incorrect if it terminated.

We have highlighted the work that directly affected the mathematical foundation of the current model of the *wp* calculus. Early pioneers in specification and verification techniques were Floyd [14] who motivated Dijkstra with his work on flow charts of a program's execution, Greif and Meyer [15], who gave a critique and tutorial on Hoare's work in 1981, Mills [16], who advocated the use of mathematical techniques to describe programming objects and concepts as early as 1975, and van Emden [17] who gave the semantics of predicate logic as a programming language in 1976. Cohen [18], Dijkstra and Scholten [19], Conway and Gries [20], and Gries [21] have all worked to develop the techniques of specification and verification using the *wp* calculus, and have presented their work as text books.

CHAPTER 4

Semantics of A Language

One of the basic tenets of our research is that programs execute. Existing work is concerned with the initial state/final state paradigm of the *wp* calculus, but we believe that much more can be said about the execution of a program than an examination of its initial state/final state pair can show. This is particularly true of programs computing partial functions because they may not terminate. We use the results of each stage of a program's execution and verify correctness from program state to program state. We use the entire sequence of a computation, so whether a computation step leads to a final state or not we can verify that one execution of the statements that comprise the step are correct.

In this chapter we define a necessary environment for our computational sequences. First we give the syntax of a programming language so we will be able to record the algorithms we use to compute partial functions. The semantics of our language are critical because we must be able to express the meaning of terminating and nonterminating computational sequences with defined results, and the meaning of nonterminating computational sequences with undefined results. We provide a denotational semantics that admits an undefined element so we can express the value of a partial function that is undefined on its input. The operational semantics of our language is the foundation for our computational sequences. To generate a computational se-

quence we need a machine. To accurately define the actions of the machine we provide operational semantics. We define a machine because we expect our programs to execute, and we use our operational semantics to formalize the meaning of a program's execution.

4.1 Syntax of the Language

We compute with the natural numbers. Our programming language has an addition operator, and a subtraction operator (monus), and uses a sequence of indexed variables to store input, output, and intermediate values. We represent individual variables in the sequence with identifiers. Our language permits Boolean expressions as guards for loops and we allow unbounded looping. We also include two operators that allow us to store a sequence of natural numbers in a variable and to retrieve the leftmost value stored in such a sequence. The numbers in the sequence are separated by a delimiter that has no value.* The programming language is similar to the language used by Sommerhalder and van Westrhenen [22]. The syntax of this language is given in Figure 4.1. Proof that this model forms a complete programming system, sufficient to express the set of all computable functions, is a standard exercise and is omitted here. See Sommerhalder and van Westrhenen for details [22].

Let G be the grammar defined by the productions in Figure 4.1, and let $L(G)$ be the set of all valid sentences that can be constructed from G . We define a program to be a quadruple $Q = (p, k, n, P)$ consisting of three natural numbers and a sequence of statements $P \in L(G)$. The identifiers that occur in the sequence P are the program variables and ultimately will represent machine registers. The numbers p, k specify that x_1, x_2, \dots, x_p and x_1, x_2, \dots, x_k are the input and output variables

*These operators are included only for expressibility and notational convenience. They add no additional computational power to our language.

$\langle \text{identifier} \rangle ::= \{x_{\langle \text{numeral} \rangle}\}$
 $\langle \text{numeral} \rangle ::= \langle \text{pos} \rangle \langle \text{numeral} \rangle \mid \langle \text{digit} \rangle$
 $\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid \dots \mid 9$
 $\langle \text{pos} \rangle ::= 1 \mid 2 \mid 3 \mid \dots \mid 9$
 $\langle \text{expression} \rangle ::= \langle \text{expression} \rangle + \langle \text{expression} \rangle \mid$
 $\quad \langle \text{expression} \rangle - \langle \text{expression} \rangle \mid (\langle \text{expression} \rangle)$
 $\quad \langle \text{identifier} \rangle \mid \langle \text{numeral} \rangle$
 $\langle \text{relop} \rangle ::= < \mid \leq \mid = \mid \neq \mid \geq \mid >$
 $\langle \text{Boolean constant} \rangle ::= \text{TRUE} \mid \text{FALSE}$
 $\langle \text{guard} \rangle ::= \langle \text{Boolean expression} \rangle$
 $\langle \text{Boolean expression} \rangle ::= \langle \text{expression} \rangle \text{ or } \langle \text{expression} \rangle \mid$
 $\quad \langle \text{expression} \rangle \text{ and } \langle \text{expression} \rangle \mid \text{not } \langle \text{expression} \rangle \mid$
 $\quad (\langle \text{expression} \rangle) \mid \langle \text{identifier} \rangle \langle \text{relop} \rangle \langle \text{identifier} \rangle \mid$
 $\quad \langle \text{identifier} \rangle \langle \text{relop} \rangle \langle \text{numeral} \rangle \mid$
 $\quad \langle \text{Boolean constant} \rangle$
 $\langle \text{assignment} \rangle ::= \langle \text{identifier} \rangle := \langle \text{expression} \rangle \mid$
 $\quad \langle \text{identifier} \rangle := \langle \text{identifier} \rangle \parallel \langle \text{identifier} \rangle \mid$
 $\quad \langle \text{identifier} \rangle := \parallel \langle \text{identifier} \rangle$
 $\langle \text{statement} \rangle ::= \langle \text{assignment} \rangle \mid \text{while } \langle \text{guard} \rangle \text{ do } \langle \text{sequence} \rangle \text{ od} \mid$
 $\quad \text{if } \langle \text{guard} \rangle \text{ then } \langle \text{sequence}_1 \rangle \text{ else } \langle \text{sequence}_2 \rangle \text{ fi}$
 $\langle \text{sequence} \rangle ::= \langle \text{statement} \rangle \mid \langle \text{sequence} \rangle ; \langle \text{statement} \rangle$
 $\langle \text{program} \rangle ::= \{(p, k, n, \langle \text{sequence} \rangle); \text{end.} \mid p \geq 0, k > 0 \text{ and } p, k \leq n\}$

Figure 4.1. Syntax of the Language.

and the number n specifies that all these variables and all program variables belong to the set $\{x_1, x_2, \dots, x_n\}$.

We require that $p \geq 0$ to express that programs may or may not have input. We require that $k > 0$ to express that programs have output. The value n expresses that we require a fixed number of variables for any program. In the syntax of this language, we often denote a sequence by $S_1; S_2; \dots; S_n$ to imply the individual statements in the sequence. We denote a *guard* by \mathcal{G} , and we often use the letter e to denote *expressions*.

4.2 Semantics of the Language

It is natural to take the meaning of a program to be the actions that a machine takes upon it. The *operational semantics* of a program uses a machine to define a language. The meaning of a program in a language is the evaluation history or computational sequence that the machine produces when it executes the program. The *denotational semantics* method maps a program directly to its meaning, called its *denotation*. The denotation is usually a mathematical value, such as a number or a function. A machine is not a part of the denotational definition; a *valuation function* maps a program directly to its meaning.

4.2.1 The Denotational Semantics of the Language

The objects we have in mind when we compute a value are the natural numbers $\mathbf{IN} = \{0, 1, 2, \dots\}$. We represent these objects as strings belonging to the set \mathbf{NAT} , where $\mathbf{NAT} = \{x \mid \langle numeral \rangle \stackrel{\pm}{\Rightarrow} x\}$. The valuation function for the set \mathbf{NAT} maps it to the set \mathbf{IN} in the expected way. We also use sequences of natural numbers separated by the delimiter $\#$, where $\#$ has no value. We denote this set $\mathcal{N} = \mathbf{NAT}(\#\mathbf{NAT})^*$. Note that $\mathbf{NAT} \subset \mathcal{N}$. Sequences containing delimiters are not used to compute values so that for a string x , if $x \in \mathcal{N} - \mathbf{NAT}$ then the valuation function returns *undefined* if computation with x is attempted. Finally, we use the set $\mathbf{IB} = \{TRUE, FALSE\}$ of Boolean values.

The objects we compute with are constructed by a finite number of applications of operators from a set of operators we designate for this purpose. The operators are applied to three distinct types of elementary objects.

1. Three types of objects:

- $\mathbf{NAT} = \{0, 1, 2, \dots, 9, 10, \dots\}$

- $\mathcal{N} = \text{NAT}(\# \text{NAT})^*$
- $\mathbb{B} = \{TRUE, FALSE\}$

2. Five operators:

- 0, a nullary operator.
- $+$, representing addition of the natural numbers.
- $\dot{-}$, (*monus*) representing subtraction of the natural numbers.
- $\|: \mathcal{N} \rightarrow \mathcal{N}$ defined by

$$\| (x, y) \triangleq x \# y.$$

$\|$ takes $x, y \in \mathcal{N}$ and returns the sequence $x \# y$.

- $\|: \mathcal{N} \rightarrow \text{NAT}$ defined by:

$$\| (x) \triangleq a, \text{ such that } a \in \text{NAT for some } ay = x \in \mathcal{N}$$

The sequence y may be empty. $\|$ takes a sequence $x \in \mathcal{N}$ and removes the leftmost natural number a .

Denotational Semantics

Denotational semantics defines a mapping from syntactical entities to the values they compute in such a way that the meaning or value of a construct is determined in terms of the meanings of its constituent parts. This is the method of Scott and Strachey [10]. In this view, a valuation function is defined that maps statements to their values. We use a variant of de Bakker's notation [5] that includes the *state* and the *state function*. A state is the condition, at any given moment, of a program's identifiers; it is an element of \mathcal{N}^n . The state function is denoted by γ , and allows us to obtain the current value of any variables used in a statement. Recall, we represent

the elements of our program state space Γ with γ . In a sense, γ retrieves the result of the valuation function applied to a statement in a particular state of the computation so that we can inspect it. The set Γ is the set of all valuations of a given program's variables. Another representation for a given $\gamma \in \Gamma$ is $\langle x_1, \dots, x_n \rangle$, the variable vector used by our programs. We let \mathcal{M} stand for the function that maps statements to their meanings: $\mathcal{M} : S \in L(G) \times \Gamma \rightarrow \Gamma$. The *valuation function* or *meaning function* $\mathcal{M}(S, \gamma) = \gamma'$, for $S \in L(G)$, reflects that the initial state γ is transformed by S into the final state γ' , where γ and γ' determine the initial and final values of the variables before and after S executes. In the recursive class \mathcal{M} is a partial function, because nontermination is a possibility. It may be that for a given γ and S there is no γ' such that $\mathcal{M}(S, \gamma) = \gamma'$.

To address the possibility of nontermination in denotational semantics it is customary to extend \mathcal{M} to a total function by adding the undefined state:

$$\mathcal{M} : S \times \Gamma \cup \{\perp\} \rightarrow \Gamma \cup \{\perp\},$$

where “ \perp ” (“bottom”) is the undefined state. Since $\perp \notin \Gamma$ and Γ contains all valuations of a program's variables, \perp is usually associated with the everywhere undefined state [10], [5], [6]. For example, in denotational semantics $\mathcal{M}(\text{while } TRUE \text{ do } S \text{ od}, \gamma) = \perp$. The extended meaning function

$$\mathcal{M} : S \times \Gamma \cup \{\perp\} \rightarrow \Gamma \cup \{\perp\}$$

is *strict* (notation: $\mathcal{M} : S \times \Gamma \cup \{\perp\} \rightarrow_s \Gamma \cup \{\perp\}$). A function is called *strict* if $f(\perp) = \perp$. That is, the value computed by a function undefined on its arguments is undefined. The translation to strictness with respect to a denotational semantics meaning function can be expressed by the acronym, *Garbage In, Garbage Out*, that

is, defined outputs cannot be obtained from undefined inputs. Strictness applies to functions for which meaningful valuation of a statement's variables may not exist because the statement represents a partial function undefined on its argument. Our extended meaning function is strict and we rely on strictness to guarantee that if a meaningful valuation is impossible in a given state, then a valid definition cannot magically be assigned by any statement. A formal discussion of strictness is given in Section 4.2.3.

The only statement in our language for which termination is an issue is the **while do od** statement. Letting ϕ stand for the result of a typical element of the set of extended meaning functions $\mathcal{M} : S \times \Gamma \cup \{\perp\} \rightarrow \Gamma \cup \{\perp\}$, we must determine a ϕ such that $\mathcal{M}(\text{while } \mathcal{G} \text{ do } S \text{ od}, \gamma) = \phi$. Intuitively by this statement we mean: repeat execution of S zero or more times as long as $\mathcal{G} = \text{TRUE}$. This description can be represented as the limit of a sequence of approximations $\phi_i, i = 0, 1, 2, \dots$, (notation: $\bigsqcup_{i=0}^{\infty} \phi_i$). The limit of a sequence of approximations is the actual function value, if it exists. For example, with $f(x) = x!$, $f(4) = 24$ is the limit and 1, 2, 6, 24 is the sequence of approximations to $f(4)$. If a limit does not exist, e.g., $f(x) = x + 1$, then the limit is undefined, but for each $i = 0, 1, 2, \dots$, $\bigsqcup_{i=0}^{\infty} \phi_i$ represents the limit of its subsequences since for each $i \geq 0$, ϕ_i is the meaning that results from executing S at most $i - 1$ times. The denotational semantics of a statement in our language is a variant of de Bakker's [5] that makes use of the state \perp and function ϕ just defined.

Definition 4.1 *The denotational semantics of $S \in L(G)$ is given by the function $\mathcal{M} : S \times \Gamma \cup \{\perp\} \rightarrow \Gamma \cup \{\perp\}$.*

$$1. \mathcal{M}(x_i := x_j + x_k, \gamma) =$$

$$\lambda \gamma. \left[\begin{array}{ll} \{ \langle x_{1\gamma}, \dots, x_{i\gamma} = x_{j\gamma} + x_{k\gamma}, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle \} & \text{if } x_j, x_k \in \mathbf{NAT} \\ \perp & \text{otherwise} \end{array} \right]$$

$x_{i\gamma} \equiv \gamma(x_i)$, the value of x_i in γ . Recall, γ represents a state, and allows

us to obtain the value of variables in a state. \mathcal{M} returns the least γ such that variable x_i obtains a new value because the arguments to addition represent natural numbers. The state that results is \perp if the arguments represent elements of $\mathcal{N} - \text{NAT}$ because variables containing delimiters are not used to compute.

$$2. \mathcal{M}(x_i := x_j - x_k, \gamma) =$$

$$\lambda. \gamma \left[\begin{array}{ll} \{ \langle x_{1\gamma}, \dots, x_{i\gamma} = x_{j\gamma} - x_{k\gamma}, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle \} & \text{if } x_j, x_k \in \text{NAT and } x_{j\gamma} \geq x_{k\gamma} \\ \{ \langle x_{1\gamma}, \dots, x_{i\gamma} = 0, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle \} & \text{if } x_j, x_k \in \text{NAT and } x_{j\gamma} < x_{k\gamma} \\ \perp & \text{otherwise} \end{array} \right]$$

If the arguments to monus represent natural numbers, then the value returned is the result of subtraction on the natural numbers. Otherwise, one or both arguments contain delimiters, so we return \perp .

$$3. \mathcal{M}(x_i := x_j, \gamma) = \lambda. \gamma \{ \langle x_{1\gamma}, \dots, x_{i\gamma} = x_{j\gamma}, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle \} \mid x_j \in \mathcal{N}$$

$$4. \mathcal{M}(x_i := c, \gamma) = \lambda. \gamma \{ \langle x_{1\gamma}, \dots, x_{i\gamma} = c, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle \}, \quad c \in \text{NAT}$$

If the right-hand side of the assignment statement is a natural number, then the variable on the left-hand side of the assignment statement takes on the value of that constant.

$$5. \mathcal{M}(S1; S2, \gamma) = \lambda. \gamma. \mathcal{M}(S2, \mathcal{M}(S1, \gamma))$$

The semicolon is used to combine adjacent, independent commands into a single command [21]. We will formalize this notion in Definition 4.3 and discuss it in depth in Chapter 4. To obtain the value of computing the whole statement $S1; S2$, we find the value of $S1$ and then use this state when we operate to obtain the value of $S2$.

$$6. \mathcal{M}(\text{if } \mathcal{G} \text{ then } S1 \text{ else } S2 \text{ fi}, \gamma) =$$

$$\lambda\gamma. \left[\begin{array}{ll} \mathcal{M}(S1, \gamma) & \text{if } \mathcal{G} \text{ is TRUE in } \gamma \\ \mathcal{M}(S2, \gamma) & \text{if } \mathcal{G} \text{ is FALSE in } \gamma \end{array} \right]$$

The decision construct returns the value of executing $S1$ if the guard is **TRUE**, i.e. if $\gamma(\mathcal{G}) = \text{TRUE}$; it returns the value of executing $S2$ if the guard is **FALSE**.

7. $\mathcal{M}(\text{while } \mathcal{G} \text{ do } S \text{ od}, \gamma) = \sqcup_{i=0}^{\infty} \phi_i(\gamma)$, where

$$\left\{ \begin{array}{l} \phi_0 = \lambda\gamma. \perp \\ \phi_{i+1} = \lambda\gamma. \text{if } \mathcal{G} \text{ then } \phi_i(\mathcal{M}(S, \gamma)) \text{ else } \gamma \text{ fi}, i = 0, 1, \dots \end{array} \right.$$

The loop has no meaning until it terminates. We obtain the value of each successive iteration by applying the loop's statements to the most recent approximation to the computation's limit as long as conditions for termination are not met. See Figure 4.2 for an example.

8. $\mathcal{M}(\text{end.}, \gamma) = \gamma$

The "end." statement does not affect the value of variables in a state.

9. $\mathcal{M}(x_i := x_j \parallel x_k, \gamma) = \lambda\gamma. \{ \langle x_{1\gamma}, \dots, x_{i\gamma} = x_{j\gamma} \parallel x_{k\gamma}, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle \}$

The concatenation operator appends the first identifier onto the left hand side of the second operator and produces a state with the updated string as one of its values.

10. $\mathcal{M}(x_i := \parallel x_j, \langle x_{1\gamma}, \dots, x_{j\gamma} = \sigma_1 \parallel \sigma_2 \parallel \dots \parallel \sigma_l, x_{j+1\gamma}, \dots, x_{n\gamma} \rangle) =$

$$\lambda\gamma. \left[\begin{array}{ll} \{ \langle x_{1\gamma}, \dots, x_{i\gamma} = \sigma_1, \dots, x_{j\gamma} = \sigma_2 \parallel \dots \parallel \sigma_l, x_{j+1\gamma}, \dots, x_{n\gamma} \rangle \} & \text{if } x_j \in \mathcal{N} \\ \perp & \text{otherwise} \end{array} \right]$$

The deconcatenation operator moves the leftmost representation of a natural number to the variable on the left hand side of the assignment operator, leav-

$\mathcal{M}(\text{while } x > 0 \text{ do } x := x - 1 \text{ od}, \gamma(< 2 >)) = (\bigsqcup_{i=0}^{\infty} \phi_i) \gamma(< 2 >) = \gamma(< 0 >)$, since

$$\begin{aligned}
 \phi_0(\gamma(< 2 >)) &= \perp \\
 \phi_1(\gamma(< 2 >)) &= \phi_0(\gamma(< 1 >)) = \perp \\
 \phi_2(\gamma(< 2 >)) &= \phi_1(\gamma(< 1 >)) = \phi_0(\gamma(< 0 >)) = \perp \\
 \phi_3(\gamma(< 2 >)) &= \phi_2(\gamma(< 1 >)) = \phi_1(\gamma(< 0 >)) = \gamma(< 0 >) \\
 &\text{and} \\
 \phi_i(\gamma(< 2 >)) &= \gamma(< 0 >), i > 3. \\
 &\text{so} \\
 (\bigsqcup_{i=0}^{\infty} \phi_i)(\gamma(< 2 >)) &= \bigsqcup_{i=0}^{\infty} \phi_i(\gamma(< 2 >)) = \bigsqcup_{i=0}^{\infty} \gamma(< 0 >) = (\gamma(< 0 >))
 \end{aligned}$$

Figure 4.2. Example of the denotational definition of whiledo od.

ing the remainder of the string in the variable on the right hand side of the assignment operator. If x_j has not legally acquired a meaningful value through execution of a previous statement, then \perp is returned.

4.2.2 The Operational Semantics of the Language

The fixed point argument that leads to our inductive proof method hinges on the computational sequences built by our operational semantics. We contend that the initial state/final state model used by the *wp* calculus is insufficient for partial functions because the moment of termination for programs computing them cannot be known or even guaranteed. Lacking the information “contained” in the final state of a computation leaves the *wp* method dependent on the partial assurance that the *wlp* conjunct can give: The computation will not be incorrect if it terminates. Consider an operating system. In such a context this means that all we can do is start the program running and wait. We may assume that “something is happening” because we see disk lights flashing and hear whirring sounds, but we cannot say anything about what is happening for certain, because we must wait for termination since the *wp* method depends on the postcondition. The *wp* method uses only the initial

state/final state predicate pair, so until we reach a final state we cannot apply the *wp* predicate transformer. Meanwhile, the operating system is managing the machine, perhaps servicing requests from users, initiating remote printing jobs, or whatever. With the ability to “examine” intermediate results of a computation via its evaluation history, we can observe all these activities, and check to see that they are executing correctly.

The concept of evaluation history or computation sequence is tied to a machine.

The Machine

We specify computations with algorithms (finite descriptions of computations over a certain data type for any given input), and represent algorithms as programs written in the language we have described. Because we are interested in the correct behavior of computations and not just in the correctness of the algorithms that specify them, we need a computer to execute algorithms and produce an evaluation history of each execution.

The machine model we use is the Random Access Machine (RAM) described by Meyer and Ritchie [4]. The RAM consists of an instruction register of arbitrary length that holds the program. The operational semantics of our language defines the operation of the machine on this program. The RAM has a Boolean test register for evaluating any *guards* the program uses, and a data memory that is unbounded and composed of registers of arbitrary (but finite) length. See Figure 4.3. Prior to execution of any program the variable vector of our machine is composed of n undefined variables. Once input is introduced, but before execution commences, the variable vector of our machine is composed of p defined input variables, with the remaining $n - p$ variables undefined.

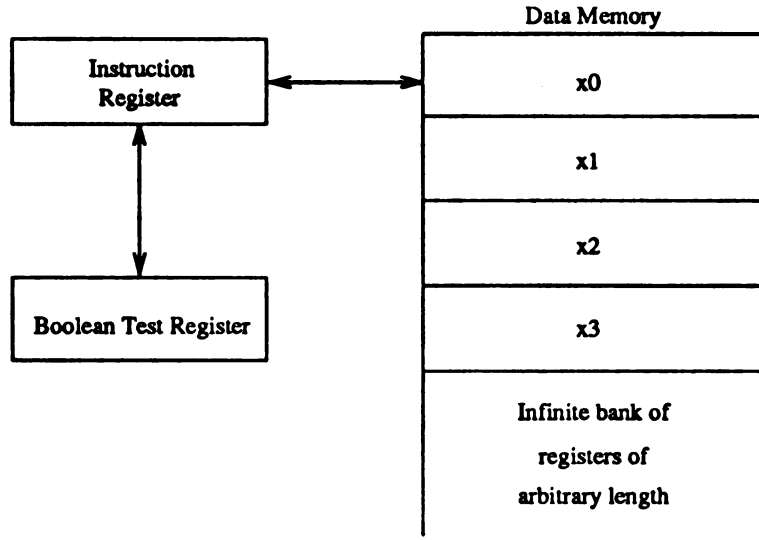


Figure 4.3. Random Access Machine (RAM).

The Operational Semantics

When we say that a program executes, we mean that our operational semantics produces a unique computation sequence that contains an evaluation history for execution of each statement, and the machine mirrors what the operational semantic function is doing. We formalize this notion of computation sequences and program meaning with operational semantics for our language.

First we formalize the way our machine implements the natural numbers. The machine consumes input and produces output. We require that all objects used in computing be finitely representable. The objects we use are the binary encodings of the natural numbers. To specify these objects, we fix a data type to formalize the data elements and the operators used by our machine.

The data type we use has three sorts. First is the set of binary strings representing the natural numbers $\text{BIN} = \{0, 1, 10, 11, 100, 101, 110, 111, \dots\}$. We normalize these strings by deleting leading zeros. This is the sort that our machine computes

values with; it is the *type of interest*. Second is the set $\mathcal{BIN} = \text{BIN}(\# \text{BIN})^*$, where syntactically $\#$ is used as a delimiter and has no semantic value in a computation. The semantic meaning of $\#$ is *undefined*. When we say that a variable in our machine vector is undefined, we mean that the position of that variable within the vector is set to $\#$. Finally, we use the set $\text{IB} = \{TRUE, FALSE\}$ of Boolean values.

1. Three sorts:

- $\text{BIN} = \{0, 1, 10, 11, 100, 101, 110, 111, \dots\}$
- $\mathcal{BIN} = \text{BIN}(\# \text{BIN})^*$
- $\text{IB} = \{TRUE, FALSE\}$

2. Ten operators:

- The functions 0 and 1, representing the elements 0 and 1 from the binary numbers.
- $+$, representing addition of the binary numbers denoted by two strings.
- $\dot{-}$, (*monus*) representing subtraction of the binary numbers denoted by two strings.
- $\|: \mathcal{BIN} \rightarrow \mathcal{BIN}$ defined by

$$\| (x, y) \triangleq x \# y.$$

- $\|: \mathcal{BIN} \rightarrow \text{BIN}$ defined by:

$$\| (x) \triangleq a, \text{ such that } a \in \text{BIN for some } ay = x \in \mathcal{BIN}$$

The string y may be empty.

- $\text{id}_{\text{BIN}}: \text{BIN} \rightarrow \text{BIN}$ and $\text{id}_{\mathcal{BIN}}: \mathcal{BIN} \rightarrow \mathcal{BIN}$, the identity functions.

- Predicate \odot_{BIN} defined by:

$$\odot_a(x) \triangleq [\text{if } x \odot a \text{ then } \text{TRUE} \text{ else } \text{FALSE}], \quad a, x \in \text{BIN}, \quad \odot \in \{<, \leq, =, \neq, \geq, >\}$$

Our data type is \mathcal{B} :

$$\mathcal{B} = \{\text{BIN}, \mathcal{BIN}, \text{IB}, \text{id}_{\text{BIN}}, \text{id}_{\mathcal{BIN}}, +, -, \parallel, \# , 0, 1, \odot_{\text{BIN}}\}$$

We express our input and output as natural numbers, $\text{IN} = \{0, 1, 2, \dots\}$. We do this for readability, because it is easier for people to think in the natural number domain than in the binary number domain. Our machine is equipped with a coding function $c : \text{IN} \rightarrow \text{BIN}$ which maps zero onto 0 and any number greater than zero onto its binary representation, with leading zeros removed. Our machine is also equipped with a decoding function $dc : \text{BIN} \rightarrow \text{IN}$. Since the function c is a bijection we define the function dc in the logical way. No decoding of elements using the $\#$ delimiter is necessary.

Definition 4.2 *A configuration δ of the machine is a pair $\delta = \langle P, \gamma \rangle$ where $P \in L(G)$ is a sequence of command statements and γ is a state in which the next command $S \in P$ can execute.*

Since our machine is digital, execution will move through discrete stages i , from a configuration, which are represented (if possible) by further configurations. These configurations will either have the form $\langle S_i, \gamma_i \rangle$ where γ_i is the state at stage i and S_i represents the remaining computation, or they will have the form $\langle \text{end.}, \gamma_i \rangle$ where i is the final stage at termination and the resulting γ_i is the machine vector in a terminal configuration.

Definition 4.3 *A functional τ over $L(G)$ maps the set of statements $L(G)$ into itself. That is, τ takes any statement $S \in L(G)$ in a given state γ as its argument and yields*

a statement $\tau(S) \in L(G)$ in a state γ' as its value.

In our language, the functional we define is the symbol “;.” For readability, we may let τ denote “;.”

Definition 4.4 A computational sequence, $\langle S_0, \gamma_0 \rangle \vdash \langle S_1, \gamma_1 \rangle \vdash \dots$, is a sequence of configurations which is infinite when no $S_i \equiv \text{end}$. A computational sequence is finite when an end. statement is encountered, i.e. $\langle S_0, \gamma \rangle, \dots, \langle \text{end.}, \gamma \rangle$. The symbols $\delta_i \vdash \delta_{i+1}$ represent the transition from one configuration to its successor under our functional that maps program statement/program state pairs to program statement/program state pairs.

Definition 4.5 The operational semantics of program statements is given by the function $\mathcal{O} : L(G) \times \Gamma \rightarrow L(G) \times \Gamma$. Let α denote a remaining sequence of program instructions that may be empty.

$$1. \mathcal{O}(x_i := x_j + x_k; \alpha, \gamma) =$$

$$\begin{cases} \langle \alpha, x_{1\gamma}, \dots, x_{i\gamma'} = x_{j\gamma} + x_{k\gamma}, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle & \text{if } x_j, x_k \in \text{BIN} \\ \langle \alpha, \#_1, \dots, \#_n \rangle & \text{otherwise} \end{cases}$$

$x_{i\gamma} \equiv \gamma(x_i)$. It represents the value of variable x_i in state γ . If the arguments to addition represented binary encodings of natural numbers, then the remaining program sequence and a new state with values reflecting the result of addition is returned. Otherwise, the remaining program sequence and a state with undefined values denoted by the valueless delimiter $\#$ is returned.

$$2. \mathcal{O}(x_i := x_j - x_k; \alpha, \gamma) =$$

$$\begin{cases} \langle \alpha, x_{1\gamma}, \dots, x_{i\gamma'} = x_{j\gamma} - x_{k\gamma}, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle & \text{if } x_j, x_k \in \text{BIN and } x_{j\gamma} \geq x_{k\gamma} \\ \langle \alpha, x_{1\gamma}, \dots, x_{i\gamma'} = 0, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle & \text{if } x_j, x_k \in \text{BIN and } x_{j\gamma} < x_{k\gamma} \\ \langle \alpha, \#_1, \dots, \#_n \rangle & \text{otherwise} \end{cases}$$

If the arguments to monus represented binary encodings of natural numbers, then the remaining program sequence and a new state with values reflecting the result of subtraction of binary encodings of natural numbers is returned. Otherwise, the remaining program sequence and a state with undefined value, denoted by the delimiter $\$$, is returned.

$$3. \mathcal{O}(x_i := x_j; \alpha, \gamma) = \langle \alpha, x_{1\gamma}, \dots, x_{i\gamma'} = x_{j\gamma}, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle, x_j \in \text{BIN}$$

We obtain the remaining program sequence and a new state with the value of x_i updated to the value of x_j , and x_j and all other variables remain as they were before the operation.

$$4. \mathcal{O}(x_i := c; \alpha, \gamma) = \langle x_{1\gamma}, \dots, x_{i\gamma'} = c, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle, c \in \text{BIN}$$

We obtain the remaining program sequence and a new state with the value of x_i updated to the binary encoding of the natural number c represents, and all other variables remain as they were before the operation.

$$5. \mathcal{O}(S1; S2; \alpha, \gamma) = \mathcal{O}(S2; \alpha, \Pi_2^2(\mathcal{O}(S1, \gamma)))$$

Π_k^p is the function $\lambda x_1 x_2 \dots x_p [x_k]$, projecting a p -dimensional vector onto its k^{th} component. We project into the computation sequence to obtain the state resulting from execution of $S1$ and use this state in our computation of $S2$.

$$6. \mathcal{O}(\text{if } \mathcal{G} \text{ then } S1 \text{ else } S2 \text{ fi}; \alpha, \gamma) =$$

$$\begin{cases} \mathcal{O}(S1; \alpha, \gamma) & \text{if } \mathcal{G} \text{ is TRUE in } \gamma \\ \mathcal{O}(S2; \alpha, \gamma) & \text{if } \mathcal{G} \text{ is FALSE in } \gamma \end{cases}$$

If the Boolean test register shows that the value of \mathcal{G} is true when tested, then we execute $S1$. We execute $S2$ if \mathcal{G} is false when tested.

7. $\mathcal{O}(\text{while } \mathcal{G} \text{ do } S \text{ od}; \alpha, \gamma) =$

$$\begin{cases} \mathcal{O}(S; \text{while } \mathcal{G} \text{ do } S \text{ od}; \alpha, \gamma) & \text{if } \mathcal{G} \text{ is TRUE in } \gamma \\ \langle \alpha, \gamma \rangle & \text{otherwise} \end{cases}$$

We extend our computation sequence one more step by unfolding the loop each time the Boolean register shows that the guard tests true. When the guard tests false, we obtain the program sequence that remains after the loop and the state that resulted from the last execution of the loop.

8. $\mathcal{O}(\text{end.}\alpha, \gamma) = \langle \emptyset, \gamma \rangle$

When an end. statement is executed, the final program state is returned and the instruction register is cleared.

9. $\mathcal{O}(x_i := x_j \parallel x_k; \alpha, \gamma) = \langle \alpha, x_{1\gamma}, \dots, x_{i\gamma'} = x_{j\gamma} \# x_{k\gamma}, x_{i+1\gamma}, \dots, x_{n\gamma} \rangle$

We obtain the remaining program sequence and a new state with variable x_i containing the string $x_j \# x_k$.

10. $\mathcal{O}(x_i := \mathbb{V} x_j; \alpha, \langle x_{1\gamma}, \dots, x_{j\gamma} = \sigma_1 \# \sigma_2 \# \dots \# \sigma_l, \dots, x_{n\gamma} \rangle) =$

$$\begin{cases} \langle \alpha, x_{1\gamma}, \dots, x_{i\gamma'} = \sigma_1, \dots, x_{j\gamma'} = \sigma_2 \# \dots \# \sigma_l, \dots, x_{n\gamma} \rangle & \text{if } x_j \in \mathcal{BIN} \\ \langle \alpha, \#_1, \dots, \#_n \rangle & \text{otherwise} \end{cases}$$

If x_j has acquired a legal value by a previous computation step, then we obtain the remaining program sequence and a new state with x_i containing the leftmost element of a string of binary representations of natural numbers separated by delimiters, and x_j containing the string that remains after removing its leftmost element. Otherwise, we return a state with coordinates set to $\#$.

Definition 4.6 *The operational meaning of a program $P \in L(G)$ with γ_0 an initial*

state, is given by:

$$\mathcal{O}^*(P, \gamma_0),$$

*where * indicates repeated composition of the operational semantic function with itself. That is, the operational semantic function continues to manipulate the sequence of program statements in the instruction register until termination.*

Thus the operational meaning of a program is the computation sequence that is induced by the functional we use operating on the program statements. From a computation sequence we can extract the functional meaning of a program.

Definition 4.7 *For any computation sequence induced by executing a program $Q = (p, k, n, P)$, the function F_Q computed by the program is defined by*

$$F_Q(\Pi_{1...k}(\gamma_0)) = \begin{cases} \Pi_{1...k}(\Pi_2^2(\mathcal{O}^j(P, \gamma_0))) & \text{if } \Pi_2^j(\mathcal{O}^j(P, \gamma_0)) = \langle \text{end.}, \gamma_j \rangle \\ \text{undefined otherwise} \end{cases}$$

Program P computes the partial function $F_Q((\Pi_{1...k}(\gamma_0)) = f(x_1, \dots, x_p)$.

At this point we give an example of how the operational semantics of a simple program builds a computation sequence, and how we can use the sequence to obtain the operational meaning of a program and the functional meaning of the program.

Example 4.1 *Let $Q = (1, 1, 1, \text{while } x > 0 \text{ do } x := x - 1 \text{ od; end.})$ and let $\gamma_0 = 2$.*

Then the operational semantics gives the computation sequence

$$\delta_0 = \langle \text{while } x_1 > 0 \text{ do } x_1 := x_1 - 1 \text{ od; end.}, 2 \rangle \vdash$$

$$\delta_1 = \langle x_1 := 2 - 1; \text{while } x_1 > 0 \text{ do } x_1 := x_1 - 1 \text{ od; end.}, 2 \rangle \vdash$$

$$\delta_2 = \langle \text{while } x_1 > 0 \text{ do } x_1 := x_1 - 1 \text{ od; end.}, 1 \rangle \vdash$$

$$\delta_3 = \langle x_1 := 1 - 1; \text{while } x_1 > 0 \text{ do } x_1 := x_1 - 1 \text{ od; end.}, 1 \rangle \vdash$$

$$\delta_4 = \langle \text{while } x_1 > 0 \text{ do } x_1 := x_1 - 1 \text{ od; end.}, 0 \rangle \vdash$$

$$\delta_5 = \langle \text{end.}, 0 \rangle$$

We can see from examination of the computation sequence of our program that its operational meaning is $\delta_0 \vdash \delta_1 \vdash \delta_2 \vdash \delta_3 \vdash \delta_4 \vdash \delta_5$. Furthermore, examination of the sixth configuration provides the additional information that the program terminates.

The functional meaning of the program can be identified by projecting onto the output variables of the program vector of the last element of the computation sequence: $F_Q(\Pi_1(\gamma_0)) = \Pi_1(\Pi_2^2(\mathcal{O}^5(P, \gamma_0))) = 0$.

Our operational semantics allows us to use a projection operator and our state valuation function γ to investigate the intermediate stages of a computational chain as a computation progresses, e.g. $\Pi_1(\Pi_2^2(\mathcal{O}^2(P, \gamma_0))) = 1$. In the theory of fixed points, projection onto the computation sequence built by our functional allows us to obtain the approximations to the function finally computed by the program. Note that the least fixed point of the function computed by our example Q is reached at the fifth configuration, i.e., $\Pi_1(\Pi_2^2(\mathcal{O}^4(P, \gamma_0))) = 0$, since for $i > 4$ projection onto our output vector returns the value 0.

4.2.3 Semantic Equivalence

Our operational semantics agree with our denotational semantics with respect to the function value computed by a program, since \perp is equivalent to *undefined*.

We support this conclusion by proving equivalence between our operational semantics and our denotational semantics, using two lemmas. The first step is a lemma that establishes that the function $\mathcal{O}(S; \alpha, \gamma)$ is strict in the same sense that the function $\mathcal{M}(S, \gamma)$ is strict, i.e., it is impossible for our machine to produce a meaningful value for a partial function that is undefined on the given input. The reason why we wish to establish strictness is continuity. It is continuity that guarantees we can obtain accurate approximations of final results at intermediate stages of a computation because the limit of a computation is predictable.

The second lemma deals with monotonicity. It says that if the execution of a

statement causes a state change, then we have moved one iteration closer to resolution. The functional “;” has produced the next stage in the computational chain.

Lemma 4.1 *Suppose the variables upon which $S \in L(G)$ depends are undefined. That is, let $x_i, \dots, x_k \in S = \#$, where $1 \leq i, \dots, k \leq n$. Represent the fact that particular undefined variables exist in S with \perp . Then $\mathcal{O}(S; \alpha, \perp) = \langle \alpha, \perp \rangle$.*

Proof: Structural induction on $S \in L(G)$.

1. $S \equiv x := e$, where e is not a natural number. $\mathcal{O}(x := e; \alpha, \perp) = \langle \alpha, \perp \rangle$. Since $\perp \notin \Gamma$, i.e., the variables in e are undefined, substitution achieved by computing an expression cannot give definition to other program variables. Substitution does not change the meaning of \perp . The two cases where the concatenation operator and the deconcatenation operator are used have the same argument.
2. $S \equiv S1; S2$. $\mathcal{O}(S1; S2; \alpha, \perp) = \mathcal{O}(S2; \alpha, \Pi_2^2(\mathcal{O}(S1, \perp)))$
 $= \mathcal{O}(S2; \alpha, \perp) = \langle \alpha, \perp \rangle$.
3. $S \equiv \text{if } \mathcal{G} \text{ then } S1 \text{ else } S2 \text{ fi}$. $\mathcal{O}(\text{if } \mathcal{G} \text{ then } S1 \text{ else } S2 \text{ fi}; \alpha, \perp) = \langle \alpha, \perp \rangle$ by definition of the operational semantics of **if fi**.
4. $S \equiv \text{while } \mathcal{G} \text{ do } S \text{ od}$. $\mathcal{O}(\text{while } \mathcal{G} \text{ do } S \text{ od}; \alpha, \perp) = \langle \alpha, \perp \rangle$. \mathcal{G} , the *guard function*, maps a Boolean expression to the set *TRUE, FALSE*.

- **Case 1:** \mathcal{G} is *TRUE* in \perp , (e.g., **while TRUE do S od**).

The operational semantic function produces the sequence
 $\mathcal{O}(\text{while TRUE do } S \text{ od}; \alpha, \perp) = \mathcal{O}(S; \text{while TRUE do } S \text{ od}; \alpha, \perp) =$
 $\mathcal{O}(\text{while TRUE do } S \text{ od}; \alpha, \perp) = \mathcal{O}(S; \text{while TRUE do } S \text{ od}; \alpha, \perp) \dots$
 which is infinite because no statement/state pair is $\langle \text{end.}, \gamma_j \rangle, j = 0, 1, \dots$ and because all $\gamma_j, j = 0, 1, \dots \in \Gamma$, the only state we can return is \perp .

- **Case 2:** \mathcal{G} is *FALSE* in \perp . This follows directly from the operational definition of the **while do od** statement.

5. $S \equiv \text{end. } \mathcal{O}(\text{end.}; \alpha, \perp) = \langle \emptyset, \perp \rangle$ by definition.

Lemma 4.2 *Let $\phi_i, i = 0, 1, \dots$, be as in the denotational definition of the while \mathcal{G} do S od statement. Then for every $\gamma \in \Gamma$ and $\gamma' \in \Gamma \cup \{\perp\}$, and for all $i \geq 0$, $\gamma' = \phi_i(\gamma)$ if and only if there is a $j, 0 \leq j < i$, such that $\mathcal{M}(S^j, \gamma) = \gamma'$. [Comment: γ' is the result of no more than $i - 1$ applications of S .]*

Proof: (“ \Rightarrow ”) Induction on i .

Basis: If $i = 0$ then $j = 0$, so there is nothing to prove.

Hypothesis: Assume $\gamma' = \phi_i(\gamma), 0 \leq j < i, i > 0$.

Show: $\gamma' = \phi_{i+1}(\gamma), 0 \leq j < i + 1$. This is $\gamma' = \lambda\gamma.\text{if } \mathcal{G} \text{ then } \phi_i(\mathcal{M}(S, \gamma)) \text{ else } \gamma \text{ fi.}$

- **Case A:** \mathcal{G} is *TRUE* in γ . Then the integer j we need is greater than 0 since we will execute S at least once. This implies that $\gamma'' = \mathcal{M}(S^{j-1}, \gamma''')$ for some γ'' and γ''' , so $\gamma'' = \phi_{i-1}(\mathcal{M}(S, \gamma'''))$. One more execution gives us $\gamma' = \phi_i(\mathcal{M}(S, \gamma'')) = \phi_{i+1}(\gamma), 0 \leq j < i + 1$.
- **Case B:** \mathcal{G} is *FALSE* in γ . If \mathcal{G} is *FALSE* then $\gamma' = \gamma, j > 0$ since we have executed S at least once because $\gamma \in \Gamma$, i.e., γ is defined. Thus for some $\gamma'', \gamma = \mathcal{M}(S^j, \gamma'') = \phi_i(S, \gamma'') = \gamma'$, which is by definition $\phi_{i+1}(S, \gamma)$, so $0 \leq j < i + 1$.

(“ \Leftarrow ”) Induction on i .

Basis: If $i = 0$, then $\phi_0 = \lambda\gamma.\perp \notin \Gamma$, so there is nothing to prove.

Hypothesis: Assume $\gamma' = \mathcal{M}(S^j, \gamma), 0 \leq j < i, i > 0$.

Show: If $\gamma' = \mathcal{M}(S^j, \gamma), 0 \leq j < i + 1$, then $\gamma' = \phi_{i+1}(\gamma) = \text{if } \mathcal{G} \text{ then } \phi_i(\mathcal{M}(S, \gamma)) \text{ else } \gamma \text{ fi.}$

T

P

st

de

En

- **Case A:** \mathcal{G} is *TRUE* in γ . Then $\gamma' = \phi_i(\mathcal{M}(S, \gamma))$ which is $\phi_i(\gamma'')$ a previous element of the computational chain. We have assumed that there is a j' such that $\gamma' = \mathcal{M}(S^{j'}, \gamma'')$. Letting $j = j' + 1$ gives us that $\gamma' = \mathcal{M}(S^j, \gamma), 0 \leq j < i + 1$.
- **Case B:** \mathcal{G} is *FALSE* in γ . If \mathcal{G} is *FALSE* then $\gamma = \phi_i(\mathcal{M}(S, \gamma''))$ for some previous element in the chain and there was a j such that $\gamma = \mathcal{M}(S^j, \gamma'')$, but this is $\gamma = \mathcal{M}(S^j, \gamma'') = \gamma' = \phi_{i+1}(\gamma), 0 \leq j < i + 1$.

Theorem 4.1 $\mathcal{O}(S; \alpha, \gamma) = \mathcal{M}(S, \gamma)$.

Proof: Let $\alpha = \emptyset$, and use structural induction on $S \in L(G)$. If S is not an iterative statement, the result follows immediately from the definitions of operational and denotational semantics. Now let $S \equiv \text{while } \mathcal{G} \text{ do } S \text{ od}$. Assume that $\mathcal{O}(S, \gamma) = \gamma'$. Either $\gamma' \in \Gamma$ or $\gamma' = \perp$.

- **Case A:** $\gamma' \in \Gamma$. Then there is a $j \geq 0$ such that $\gamma' = \mathcal{M}(S^j, \gamma)$. Let $\phi_0 = \lambda\gamma. \perp$ and let $\phi_{i+1} = \lambda\gamma. \text{if } \mathcal{G} \text{ then } \phi_i(\mathcal{M}(S, \gamma)) \text{ else } \gamma \text{ fi}, i = 0, 1, \dots$. The denotational definition of S is $\mathcal{M}(S, \gamma) = (\bigsqcup_{i=0}^{\infty} \phi_i)(\gamma)$. By Lemma 4.2, $\gamma' = \phi_i(\gamma), 0 \leq j < i + 1$, so $\gamma' = \bigsqcup_{i=0}^{\infty} \phi_i(\gamma) = (\bigsqcup_{i=0}^{\infty} \phi_i)(\gamma) = \mathcal{M}(S, \gamma)$.
- **Case B:** $\gamma' = \perp$. By Lemma 4.1 we have that our computation sequence is infinite and for all $k = 0, 1, \dots$, $\phi'_k(\gamma) = \perp$, since if for some $i \geq 0$, $\phi'_i(\gamma) = \gamma'' \in \Gamma$, by Lemma 4.2 there would exist a $j, 0 \leq j < i$ such that $\gamma'' = \mathcal{M}(S^j, \gamma)$ and this would give us that $\mathcal{O}(S, \gamma) = \gamma'' \in \Gamma$ and $\mathcal{O}(S, \gamma) = \perp$, a contradiction. Therefore for all $k \geq 0$ $\phi_k(\gamma) = \perp$ and this is the same as $\mathcal{M}(S, \gamma) = \perp$.

CHAPTER 5

Mathematical Foundation for Reasoning About Nonterminating Computations

In this chapter we give the mathematical foundation we use to reason logically about unbounded and nonterminating computations. We have already given our machine and its language, and we have defined formally what we mean when we say that program statements execute. Operationally this meaning is the computational sequence induced by our higher order function from program statements to program statements. The computational sequence is the vehicle we use to reason about a computation.

There are two types of postconditions for loops:

1. *Descriptive* postconditions define non-empty sets of program states. The Dijkstra/Gries [2], [21] method requires proof of termination by manipulation of an *a priori* upper bound. In fact, the requirement of a known upper bound limits this method to the primitive recursive class. Therefore, descriptive postconditions can be written for every loop.

2. *Nondescriptive* postconditions, i.e., *FALSE*, define empty sets of program states. Nelson [6] allows nontermination. In his generalization of Dijkstra's *wp* calculus, Nelson addresses recursively specified programs but provides a method for showing only overall partial correctness of these partial functions. In Nelson's method the postcondition of a nonterminating recursion is *FALSE* because a final state cannot be reached, i.e., the set of final states is empty. Like Nelson, we consider *FALSE* an appropriate postcondition for a nonterminating loop. Unlike Nelson we use much more descriptive predicate logic sentences for nonterminating computations than *FALSE*. We eschew the initial state/final state paradigm in these computations because we model them as chains composed of intermediate stages, where each stage is an approximation to the limit of the computation, so each stage has a predicate logic sentence that is an approximation to the goal the loop's designer intended.

Gries has said that "loops are designed for specific purposes—to establish the truth of one particular postcondition" [21]. For total recursive functions the "specific purpose" of a loop and the postcondition of the loop are identical. The "specific purpose" or goal of the loop is the machine state at termination, and this state can be described logically. Postconditions are assertions that hold about the output of a loop after the loop terminates. Outside the total recursive class, the goal of a loop can still be expressed logically, but it may not be the same as the postcondition of the loop. The goals of some loops are achieved independent of termination, the loops produce output and so require output assertions, but these assertions cannot be considered "postconditions" using the accepted definition of the word.

Loops that are "defined on their input" are loops that accomplish the specific purpose of their designer whether they terminate or not. An example of a loop that is defined on its input and terminates is Ackermann's function, a total recursive function. Another example of such a loop is the primitive recursive function for

factorial. An example of a loop that is defined on its input and does not terminate is an operating system service procedure that loops until it receives a request, then fulfills the request and returns to its waiting state. The example of the nonterminating operating system service procedure highlights the need for a logical description of a loop's goal independent of the postcondition. Since the loop does not terminate its postcondition must be *FALSE*, but the specific purpose of such a loop is certainly more descriptive than that.

We are interested here in those partial functions with *descriptive goals*. These are functions defined on their input. When these functions are computed by programs, the purpose of the program is to reach a state whose logical description is equivalent to the designer's specific purpose. For terminating loops the conjunction of this logical expression, the loop invariant, and a false guard implies a descriptive postcondition. For nonterminating loops defined on their input we can use the conjunction of the logical description of the loop's goal and the invariant to prove that each iteration of the loop is correct.

To achieve this goal we show that programs that use loops induce computational sequences that are chains. The existence of computational chains guarantees that we can obtain accurate information about the computation at intermediate stages (chains have the property of monotonicity). Chains also guarantee that the information we obtain is an approximation of the final result because the limit of a computation is predictable (chains have the property of continuity). From this we show that defined loops have descriptive goals that can be used with the loop invariant to prove the loop is incrementally correct.

It follows from a result of Kleene that all chains have least upper bounds, and that these are fixed points of the computation [7]. Fixed points can be viewed as "closed form" expressions of recursively specified programs [23], [6], or as approximations to the solution of repetitive computations [5], [24]. The program state that occurs at the

least upper bound of a computational chain has a logical sentence that defines the specific purpose of the loop because it is the first true approximation to the function computed by the loop; all other approximations are extensions of it [7].

In order to establish the existence of computational chains induced by programs with loops we define a functional that maps program statements to program statements. We prove that the functional is continuous for all noniterative statements, that the functional is continuous over one application of our looping construct, and that the functional is continuous over multiple applications.

To impose the structure we need for discussing continuity and for establishing the correspondence we need between our mathematical foundation and the predicate transformer of our proof method, we show that the state space composed of all possible variable valuations of our programs is a complete lattice. Like Nelson [6] we provide for the possibility of a program mapping a defined initial state to an undefined set of final states. This is the mapping that occurs in the case of a nonterminating computation, and it is the justification for the *FALSE* predicate logic postcondition of a loop which we use as our least element. Every continuous chain induced by a program within this lattice has a least upper bound, and this gives us our top element [7].

The organization of our lattice depends on a partial ordering relation “more defined,” imposed by our program statements which we, as others (Dijkstra [2], [1], Hehner [8], Gries [21], Nelson [6]) view as commands mapping states to states. We view the relation as a means of measuring the information content of the states our commands map to, and use it to describe the quality of the approximation to a solution each state in a computational chain represents.

5.1 The Semi-Lattice of Computations

More-defined is a relational on the set of commands on a set of program states Γ .

Definition 5.1 *Let C and C' be two commands on Γ . We say C is more-defined than C' , notation: $C' \prec C$, if and only if*

1. $\text{dom}(C') \subseteq \text{dom}(C)$,
2. $\forall \gamma \in \text{dom}(C'), \gamma.C \subseteq \gamma.C'$.

Proposition 5.1 *\prec is a partial ordering.*

Proof:

- $C \prec C$. Therefore \prec is reflexive.
- Let C and C' be two commands such that $C' \prec C$ and $C \prec C'$. From $\text{dom}(C') \subseteq \text{dom}(C)$ and $\text{dom}(C) \subseteq \text{dom}(C')$ we have that $\text{dom}(C) = \text{dom}(C')$. For all $\gamma \in \text{dom}(C)$, $\gamma.C \subseteq \gamma.C'$ and $\gamma.C' \subseteq \gamma.C$, so $\gamma.C = \gamma.C'$, so $C = C'$. Therefore \prec is antisymmetric.
- Let C, C' and C'' be three commands such that $C'' \prec C'$ and $C' \prec C$. Then $\text{dom}(C') \subseteq \text{dom}(C)$ and $\text{dom}(C'') \subseteq \text{dom}(C')$, so $\text{dom}(C'') \subseteq \text{dom}(C)$. Let γ be an element of $\text{dom}(C'')$ so it is in $\text{dom}(C')$. From $\gamma.C \subseteq \gamma.C'$ and $\gamma.C' \subseteq \gamma.C''$ we have that $\gamma.C \subseteq \gamma.C''$. Therefore, \prec is transitive. \square

We want to show that this partially ordered structure is a lattice so we can use the lattice properties of *more-defined* to measure the information content of elements in a computational chain. Given two commands C and C' on Γ , we are interested in the existence of $\text{lub}(C, C')$. The least upper bound of two commands represents the sum of the quantities of input and output information contained in them. Those quantities can only be summed if the two relations defined by the commands have

at least one element in common. The property of monotonicity guarantees that the application of successive commands in a computational chain does not disturb the established ordering relation, *i.e.*, if two commands have a common initial state then the application of the commands must yield a state that is *more-defined* than the initial state. In this sense the application of commands in a computational chain yields results that are approximations to their mutual least upper bound; they are *consistent* in maintaining the ordering relation *more-defined* on the lattice.

Consistency expresses the property of monotonicity and is a necessary and sufficient condition for the existence of a least upper bound in a computational chain induced by commands in a lattice ordered by *more-defined*. Mili, Boudriga, and Mili [25] formalized this.

Proposition 5.2 *Two commands C and C' satisfy the consistency condition*

$$\text{dom}(C \cap C') = \text{dom}(C) \cap \text{dom}(C'),$$

if and only if C and C' have a unique least upper bound.

Mili *et. al.* [25] express the unique least upper bound as the union of the states in C and in C' , unioned with the intersection of states in C and C' :

$$\text{lub}(C, C') = (\text{id}_\Gamma(\text{dom}(C) - \text{dom}(C'))C) \cup (\text{id}_\Gamma(\text{dom}(C') - \text{dom}(C))C') \cup (C \cap C').$$

This expression is suitable for the commands of our language within a lattice ordered by *more-defined*. It is an upper bound because it is equal to $\text{dom}(C) \cup \text{dom}(C')$, therefore larger than both $\text{dom}(C)$ and $\text{dom}(C')$, condition one of our definition of *more-defined*. Informally, the expression is minimal because for any other possible upper bound, say C'' , $\text{dom}(C) \subseteq \text{dom}(C'')$ and $\text{dom}(C') \subseteq \text{dom}(C'')$. Thus, $\text{dom}(C) \cup \text{dom}(C') \subseteq \text{dom}(C'')$. Because of the expression of $\text{lub}(C, C')$, we have that

$\text{dom}(\text{lub}(C, C')) \subseteq \text{dom}(C'')$, so C'' is more-defined than $\text{lub}(C, C')$.

In a sense, the least upper bound of two commands in a lattice organized by the relation *more-defined* can be compared to the *weakest liberal precondition* of the *wp* calculus, because both address the expected result of program execution.

The greatest lower bound of two commands represents the amount of redundancy between the quantities of information they carry. Greatest lower bounds provide common initial information for pairs of commands. In a sense, $\text{glb}(C, C')$ for two commands C and C' on Γ in a lattice organized by the relation *more-defined* parallels the *weakest precondition* on a set of states in the *wp* calculus because both define the largest set of initial states from which the two commands can execute correctly. The greatest lower bound of any pair of commands always exists in this lattice.

Proposition 5.3 *Mili et al [25]. Any pair of commands C and C' have a greatest lower bound, which is given by the expression*

$$\text{glb}(C, C') = \text{id}_{\text{dom}(C) \cap \text{dom}(C')}(C \cup C').$$

Since we want to map *FALSE* to some state, like Nelson [6] and Mili [25], we define the *everywhere undefined* relation, a relation capable of mapping defined states to no states:

$$C_\emptyset = \Gamma \times \emptyset.$$

C_\emptyset will be the minimum in our lattice.

For this relation we extend the definitions of domain and image set to have the following identities:

- $\text{dom}(C_\emptyset) = \Gamma,$
- $\forall \gamma, \gamma.C_\emptyset = \emptyset.$

As a relation C_\emptyset is more-defined than all other relations because it has the largest

domain which is condition one of our definition of more-defined. According to condition two of our definition of more-defined, we see that any other relation is more-defined than C_\emptyset , since for any other relation C , $\forall \gamma \in \text{dom}(C)$, $\gamma.C_\emptyset \subseteq \gamma.C$.

In summary, the set of commands viewed as binary relations is partially ordered by the relational *more-defined*, and each pair of commands has a greatest lower bound. Under a condition of mutual consistency, each pair of commands has a least upper bound. This partial ordering defines a lattice on the state space.

Next we investigate the requirements for mutual consistency among commands, and show that computations can be represented as ascending chains in the lattice of our state space, and that in each such chain mutual consistency exists among all the commands used to induce it. We show that because every ascending chain has a least upper bound, our lattice is actually complete.

5.2 Fixed Points

Our programs $Q = (p, k, n, P)$ represent n -ary partial functions, $n \geq 0$, from vectors of natural numbers to vectors of natural numbers. That is, for every element of \mathcal{N}^n , $f(x_1, x_2, \dots, x_n)$ either terminates and yields some element of \mathcal{N}^n or results in a computational sequence of infinite length which does not produce a final value. Guard functions are n -ary partial predicates that map \mathcal{N}^n into $\{TRUE, FALSE\}$.

Because we are dealing with partial functions and also because sequencing is an element of our syntax it is possible that we may attempt to compute functions undefined on some x_i , $1 \leq i \leq n$. We use ω to represent variables for which definition is impossible or no value is given, and so we extend our domains \mathcal{N}^n and \mathbf{IB} to $\mathcal{N}^n \cup \{\omega\}$ and $\mathbf{IB} \cup \{\omega\}$. We denote these extended domains $(\mathcal{N}_\omega)^n$ and \mathbf{IB}_ω .

5.2.1 Interpretation of Fixed Points

The operational semantics of our loops is based on a continuous functional operating on program statements whose variables obtain their valuation from our extended domain $(\mathcal{N}_\omega)^n$. Our functional is “;” which we denote by the symbol τ for readability.

In Chapter 4 the operational semantic function relied on τ to distinguish the next executable statement from the sequence of statements in the instruction register. Recall, from Section 4.2.2, $\mathcal{O}(S; \alpha, \gamma) = \langle \alpha, \gamma \rangle$, for α a possibly empty sequence; τ is functionally defined by this statement and performs a task that parallels the operational meaning of “;.”

Once we give valuation to the variables of our commands, the commands become specific partial functions of the variables they manipulate. Other program variables not appearing in a command are not affected by that command. We view specifically valued commands as maps from program states to program states. That is, all of our commands are functions and belong to the set $\{(\mathcal{N}_\omega)^n \rightarrow (\mathcal{N}_\omega)^n\}$. We will prove that these functions are monotonic.

The functional τ maps a program statement to its logical successor in a program. If the function is atomic, *i.e.*, not iterative, then its logical successor is the next sequential statement of the stored program. τ is clearly a continuous functional for noniterative functions. If the function τ operates on is iterative, then the result τ maps to depends on the condition of the guard. We will prove that τ is continuous for one application of our iterative functions, *i.e.*, τ correctly maps the sequence of statements within the scope of a while do od statement when the guard of the statement becomes false. We will prove that τ is continuous for multiple applications of our iterative functions, *i.e.*, τ correctly maps the last executable statement of a looping construct to the first executable statement when the statements in the loop have executed and the guard of the loop tests true. With the proof of continuity we

ensure that whether we take the least upper bound of a chain τ has built or apply τ to the least upper bound of such a chain we obtain identical results, because continuity preserves limits. That is, because our program statements are monotonic and because our functional τ is continuous, the computational sequences that result are chains and τ takes us to the statement in each chain that computes the least upper bound of the chain. The least upper bound of a chain represents the limit of the computation.

Definition 5.2 *The functional τ maps the set of partial functions into itself, $\tau : \{(\mathcal{N}_\omega)^n \rightarrow (\mathcal{N}_\omega)^n\} \rightarrow \{(\mathcal{N}_\omega)^n \rightarrow (\mathcal{N}_\omega)^n\}$. That is, for a program $P = S; P'$ in $L(G)$ where S is a statement in $L(G)$ and P' is a program segment in $L(G)$, (we view P , S , and P' as partial functions f , f_1 , and f_2 respectively in $\{(\mathcal{N}_\omega)^n \rightarrow (\mathcal{N}_\omega)^n\}$),*

$$\tau(P) = P'.$$

This mapping is defined explicitly in Section 4.2.2.

Kleene's First Recursion Theorem [7] states that every algorithmic description of a partial function can be defined in terms of a partial function and its arguments.

Theorem 5.4 First Recursion Theorem[Kleene]

For any $n \geq 0$, let $F(\xi; x_1, \dots, x_n)$ be a partial recursive functional, in which the function variable ξ ranges over partial functions of n variables. Then the equation

$$(\xi; x_1, \dots, x_n) = F(\xi; x_1, \dots, x_n)$$

has a partial recursive solution ϕ for ξ such that any solution ϕ' for ξ is an extension of ϕ .

This theorem says that every continuous functional τ has a least fixed point that is actually the least upper bound of the computational chain induced by the functional.

What this theorem means to us is that every value computed by a partial function is computed by some program, and furthermore, that of all the programs capable of computing that value (there are an infinite number of such programs), there is one that does the job with the least amount of information (or as Kleene phrases it: "...on the smallest range of definition," i.e., that subset of the domain where the function is total).

One way of describing fixed points is to say that the functional τ maps a function to itself (notation $\tau(f) = f$). This is the " $f(x) = x$ " view of Davis [24] who describes an approximation relation that relates approximations of function values to the actual function value, and also of Scott and Stoy [10]. When we view τ in this way we consider not so much the mappings from statement to statement (as when we are building computational chains), but we see the chain of statements as code that is itself a function that computes a value when it is executed. In this view, τ maps the code segment to the value it actually computes.

Nelson [6] and Manna [23] take the view that the fixed point produces the "best" definition of a recursively specified function. For example, consider the recursive specification of the factorial function:

$$\begin{array}{ll}
 f(x) \text{ is defined as: } & \text{if } x = 0 \text{ then } 1 \text{ else } x * f(x - 1) \\
 (1) & \equiv x! \\
 & (2)
 \end{array}$$

In (1) we have the recursive specification for a program. In (2) we have the "closed form expression," a non-recursive representation of the code. When we relate this equation to Kleene's theorem, we see that (1) is the algorithmic description of the function, and (2) is the the primitive recursive function that computes it. $x!$ is the least fixed point of this recursive specification because it exactly defines the function $f(x)$ and it is minimal because it is defined for all $0 \leq x < x + 1$, and undefined

everywhere else. $x!$ has the “smallest range of definition.”

In this latter view, fixed points are used to describe the values we intend our functions to compute. The translation to loops (that are not recursively specified) is that the specification of a loop (the right-hand side of Kleene’s equation) is the “specific purpose” or goal of the loop. As discussed in the introduction to this chapter, for total and primitive recursive functions this specific purpose is the postcondition. For partial recursive functions which may have postcondition *FALSE*, we say the specification of the loop is a logical description of the goal of the loop. In both cases, the conjunction of the loop invariant and the goal of the loop allows us to verify the correctness of each iteration of the loop.

We view the fixed point of a loop as the specification of the loop because it describes the value we expect to compute. Consider the equation for a loop:

$$\begin{aligned} \text{while}(x) \text{ is defined as : } \text{while } \mathcal{G} \text{ do } S \text{ od} &\equiv R \\ (1) &\qquad (2) \end{aligned}$$

Notice that (1) is Kleene’s algorithmic description, or “formula” for computing and (2) is a description of the value we expect the loop to compute. Our continuous functional τ maps the computational chain it induces to the least upper bound of that chain, and this least upper bound is a program state that has logical description R when the loop is correct. When the program executes there is an $i, i \geq 0$, such that the i^{th} iteration of while do od on a variable x , denoted $\text{while}^i(x)$, computes our intended value, and for all $k > i$, $\text{while}^i(x)$ does it with the least amount of information.

A good example of this is a program that computes the i^{th} prime number in ascending order. (We assume there is a macro that tests whether a number is prime by directing a search for divisors within the range $2 \leq i \leq \text{number} - 2$, returning 1 if no divisor is found, because it is the while loop we are interested in for now.) By

“macro” we mean that there is a body of code that can be substituted inline for the statement $isprime := Prime(y)$. We formalize this notion of macro in Section 6.1, and present the macro in Section 7.3.

Program Primes

```

 $z := 0;$ 
 $y := 0;$ 
while  $y \geq 0$  do
   $y := y + 1;$ 
   $isprime := Prime(y);$ 
  if  $isprime = 1$  then
     $p := y;$ 
     $z := z + 1;$ 
  fi
od

```

The least fixed point of the entire program is the undefined element because the loop that computes it is nonterminating by design, so we can never match a descriptive postcondition (we can never reach the postcondition, so we expect no final value). Thus, the established postcondition of this loop must be *FALSE*.

This corresponds to the denotational semantics we have defined which say that the meaning of a nonterminating loop is “everywhere undefined” or \perp .

This also corresponds to our operational semantics because an infinite computational sequence cannot produce final results. The infinite computational sequence is, however, a concatenation of computational subsequences each of which is a chain as we will show in Section 5.3. Since every subsequence has a least upper bound which is its fixed point, and these fixed points are not the same as the least fixed point of the entire sequence, we have an infinite number of intermediate approximations to the set of prime numbers.

5.3 Theory of Fixed Points

Our primary objective is to show that our functional τ induces computational chains within our state space, and that each such chain has a least upper bound, giving us a complete lattice. First we introduce the property of monotonicity and show that all of our program statements have this property. We then formally define the properties of monotonicity and continuity for functionals and show that our functional has these properties for single and multiple applications of a loop. We also show that τ is continuous for all of our noniterative statements. Given that our program statements are monotonic and that our functional is both monotonic and continuous, we are assured that our computational sequences are chains. From this we invoke Kleene's First Recursion Theorem that guarantees all of our chains have least upper bounds.

5.3.1 Monotonic Functions

We wish to extend our notation for the relation *more-defined* to states in the logical (but consistent) way on our extended domain $(\mathcal{N}_\omega)^n$. We let

1. $\vec{x} \prec \vec{x}$ for all $\vec{x} \in \mathcal{N}^n$.
2. $\vec{y} \prec \vec{x}$ for all $\vec{y} \in (\mathcal{N}_\omega)^n - \mathcal{N}^n$ and all $\vec{x} \in \mathcal{N}^n$.
3. $\vec{x} \prec \vec{y}$ and $\vec{y} \prec \vec{x}$ for all $x, y \in \mathcal{N}^n$.

This ordering has nothing to do with the natural ordering \leq on the natural numbers.

Definition 5.3 Monotonic Functions

An n -ary function f from $(\mathcal{N}_\omega)^n$ into $(\mathcal{N}_\omega)^n$ is said to be *monotonic* if $(x_1, \dots, x_n) \prec (y_1, \dots, y_n)$ implies $f(x_1, \dots, x_n) \prec f(y_1, \dots, y_n)$ for all $(x_1, \dots, x_n), (y_1, \dots, y_n) \in (\mathcal{N}_\omega)^n$.

By this definition we require that whenever the “input” vector \vec{y} is more-defined than or equal to the “input” vector \vec{x} , the “output” vector $f(\vec{y})$ is more defined than or

equal to the “output” vector $f(\vec{x})$. The class of all monotonic functions mapping $(\mathcal{N}_\omega)^n$ into $(\mathcal{N}_\omega)^n$ is denoted by $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$.

5.3.2 Properties of Monotonic Functions

The following properties hold for monotonic functions:

1. If f is a unary function, $f : \mathcal{N}_\omega \rightarrow \mathcal{N}_\omega$, then f is monotonic if and only if either
 - $f(\omega)$ is ω or
 - $f(x)$ is c for some constant $c \in \mathcal{N}$ and all $x \in \mathcal{N}_\omega$.
2. For $n \geq 2$, if an n -ary function $f, f : (\mathcal{N}_\omega)^n \rightarrow (\mathcal{N}_\omega)^n$, is monotonic then either
 - $f(\omega_1, \dots, \omega_n)$ is ω or
 - $f(x_1, \dots, x_n)$ is c for some constant $\vec{c} \in (\mathcal{N}_\omega)^n$ and for all $\vec{x} \in (\mathcal{N}_\omega)^n$.

Monotonicity will guarantee that our results will be predictable from one iteration to another. We rely on the property of *natural extension* to ensure monotonicity.

Definition 5.4 *An n -ary function f mapping $(\mathcal{N}_\omega)^n$ into $(\mathcal{N}_\omega)^n$ is said to be naturally extended if $f(x_1, \dots, x_n) = \omega$ whenever at least one of the x_i 's is ω .*

Natural extension assures us that no undefined computation sequence can map to a meaningful value. Natural extension also ensures that functions undefined on the variables they manipulate cannot produce defined results.

Example 5.1 *Consider the extension of the decision construct in the following way:*

- if *TRUE* then $S1$ else ω $\text{fi}(\vec{x})$ is $S1(\vec{x})$. When the guard tests true we expect $S1$ to be defined and will execute it, regardless of the definition of $S2$.
- if *FALSE* then ω else $S2$ $\text{fi}(\vec{x})$ is $S2(\vec{x})$. When the guard tests false, we expect $S2$ to be defined and will execute it, regardless of the definition of $S1$.

- if ω then $S1$ else $S2$ $\text{fi}(\vec{x})$ is ω . When the guard is undefined, we cannot execute $S1$ or $S2$ regardless of their definition. *

The definition of natural extension allows us to apply the Natural Extension Lemma of Manna [23]:

Lemma 5.1 *Every naturally extended function is monotonic.*

We have given the operational definition of our functions in Chapter 4. Now we show that they preserve monotonicity because we can naturally extend them.

Definition 5.5 *Natural Extension of Commands.*

- The assignment statement naturally extended to yield the value ω whenever its right hand side is ω is monotonic since we have $\omega \prec \omega$.
- The decision statement mapping $\{TRUE, FALSE\} \times \mathcal{N}^n$ into \mathcal{N}^n is defined for any $S1, S2 \in L(G)$ as follows:

$$\text{if } \mathcal{G} \text{ then } S1 \text{ else } S2 \text{ fi}(\vec{x}) = \begin{cases} S1(\vec{x}) & \text{if } \mathcal{G}(\vec{x}) = TRUE \\ S2(\vec{x}) & \text{if } \mathcal{G}(\vec{x}) = FALSE \end{cases}$$

This function can be naturally extended to a monotonic function mapping $\mathbf{IB}_\omega \times (\mathcal{N}_\omega)^n$ into $(\mathcal{N}_\omega)^n$ by letting, for any $S1, S2 \in L(G)$,

$$\text{if } \mathcal{G} \text{ then } S1 \text{ else } S2 \text{ fi}(\vec{x}) = \begin{cases} S1(\vec{x}) & \text{if } \mathcal{G}(\vec{x}) = TRUE \\ S2(\vec{x}) & \text{if } \mathcal{G}(\vec{x}) = FALSE \\ \omega & \text{if } \mathcal{G}(\vec{x}) = \omega \end{cases}$$

*While this function is not the natural extension of if fi , we can show it is monotonic by showing that for input vectors \vec{x} and \vec{x}' whenever $\vec{x} \prec \vec{x}'$, $(\text{if } \mathcal{G} \text{ then } S1 \text{ else } S2 \text{ fi})(\vec{x}) \prec (\text{if } \mathcal{G}' \text{ then } S1' \text{ else } S2' \text{ fi})(\vec{x}')$. Let $\vec{x} \prec \vec{x}'$ and assume $\mathcal{G} = TRUE$. Then $\text{if fi}(\vec{x}) = S1(\vec{x}) \prec S1(\vec{x}') = \text{if fi}(\vec{x}')$. Let $\vec{x} \prec \vec{x}'$ and assume $\mathcal{G} = FALSE$. Then $\text{if fi}(\vec{x}) = S2(\vec{x}) \prec S2(\vec{x}') = \text{if fi}(\vec{x}')$. Let $\vec{x} \prec \vec{x}'$ and assume $\mathcal{G} = \omega$. Then $\text{if fi}(\vec{x}) = \omega \prec \omega = \text{if fi}(\vec{x}')$.

- *The while do od function mapping $\{TRUE, FALSE\} \times \mathcal{N}^n$ into \mathcal{N}^n is defined for any $S \in L(G)$ as follows:*

$$\text{while } \mathcal{G} \text{ do } S \text{ od}(\vec{x}) = \begin{cases} S(\vec{x}) & \text{if } \mathcal{G}(\vec{x}) = TRUE \\ \vec{x} & \text{if } \mathcal{G}(\vec{x}) = FALSE \end{cases}$$

We extend this naturally into a monotonic function mapping $\mathbb{B}_\omega \times (\mathcal{N}_\omega)^n$ into $(\mathcal{N}_\omega)^n$ for any $S \in L(G)$ by letting

$$\text{while } \mathcal{G} \text{ do } S \text{ od}(\vec{x}) = \begin{cases} S(\vec{x}) & \text{if } \mathcal{G}(\vec{x}) = TRUE \\ \vec{x} & \text{if } \mathcal{G}(\vec{x}) = FALSE \\ \omega & \text{if } \mathcal{G}(\vec{x}) = \omega \end{cases}$$

Monotonicity is an important property for commands as units and also for commands composed into sequences. Composition allows sequences of functions to be defined in terms of simpler functions. If f is a function from $(\mathcal{N}_\omega)^n$ into $(\mathcal{N}_\omega)^n$ and g is a function from $(\mathcal{N}_\omega)^n$ into $(\mathcal{N}_\omega)^n$, then the *composition of f and g* is a function from $(\mathcal{N}_\omega)^n$ into $(\mathcal{N}_\omega)^n$ defined by $g(f(\vec{x}))$ for every \vec{x} in $(\mathcal{N}_\omega)^n$. If f and g are monotonic functions, then so is their composition since if f and g are monotonic, and if $\vec{x} \prec \vec{x}'$ then $f(\vec{x}) \prec f(\vec{x}')$, which implies $g(f(\vec{x})) \prec g(f(\vec{x}'))$ for all $\vec{x}, \vec{x}' \in (\mathcal{N}_\omega)^n$. So the composition of f and g is also a monotonic function.

5.3.3 Least Upper Bound

We are using our functional syntactically to build computation sequences, and semantically as a vehicle to produce a computation sequence. We view commands as partial functions naturally extended to total functions. We restate “ \prec ” in terms of functions.

Definition 5.6 Let $f, g \in \{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$. We say that g is more-defined than f , ($f \prec g$), if $f(\vec{x}) \prec g(\vec{x})$ for all $\vec{x} \in (\mathcal{N}_\omega)^n$. This relation is a partial ordering on $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$ since we have $C_\emptyset \prec f$ for any $f \in \{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$.

Definition 5.7 Let $f, g \in \{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$. We say that f is equal to g , ($f = g$), if $f(\vec{x}) = g(\vec{x})$ for all $\vec{x} \in (\mathcal{N}_\omega)^n$.

Proposition 5.5 $f = g$ if and only if $f \prec g$ and $g \prec f$.

If $f = g$ then $\text{dom}(f) \subseteq \text{dom}(g)$ and $\text{dom}(g) \subseteq \text{dom}(f)$, and for all $\gamma \in \text{dom}(f)$, $\gamma.g = \gamma.f$. Now suppose $f \prec g$ and $g \prec f$. Then $\text{dom}(f) \subseteq \text{dom}(g)$ and $\text{dom}(g) \subseteq \text{dom}(f)$, so $\text{dom}(f) = \text{dom}(g)$. Since $g \prec f$, for $\gamma \in \text{dom}(g)$, $\gamma.f = \gamma.g$. But since $f \prec g$, for $\gamma \in \text{dom}(f)$, $\gamma.g = \gamma.f$. So $f = g$. \square

Definition 5.8 Let f_0, f_1, f_2, \dots , be a sequence of functions in $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$. We denote this sequence by $\{f_i\}$. Then $\{f_i\}$ is called a chain if $f_0 \prec f_1 \prec f_2 \prec \dots$.

Definition 5.9 Let $\{f_i\}$ be a sequence of functions in $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$, and let $f \in \{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$. We say that f is an upper bound of $\{f_i\}$ if $f_i \prec f$ for every $i \geq 0$. Additionally, if $f \prec g$ for every upper bound g of $\{f_i\}$, then f is called the least upper bound of $\{f_i\}$, denoted $\text{lub}\{f_i\}$.

The least upper bound of $\{f_i\}$, if it exists, is unique since if both f and g are least upper bounds of $\{f_i\}$, then $f \prec g$ and $g \prec f$ so $f = g$.

Lemma 5.2 Mili [25] Every chain $\{f_i\}$ has a least upper bound.

The least upper bound of a computational chain is the fixed point of the computation [7]. In our model the fixed point is the first state in the computation where the specification is met. In the next section we prove that our computational sequences are chains.

5.3.4 “;” When Viewed As A Functional Is Continuous

Our *functional* τ over $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$ maps the set of functions $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$ into itself. That is, τ takes any monotonic function f defined in a given state γ as its argument and yields a monotonic function $\tau(f)$ defined in state γ' as its value.

The properties of *monotonicity* and *continuity* are important for τ . Monotonicity and continuity guarantee that we can obtain accurate information about the computation at intermediate stages. Monotonicity ensures that τ accurately maps a program statement to its logical successor. Continuity guarantees that this mapping is correct over an entire computational sequence.

Definition 5.10 Properties of the functional τ

1. τ , a functional over $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$, is said to be *monotonic* if $f \prec g$ implies $\tau(f) \prec \tau(g)$ for all $f, g \in \{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$.
2. The monotonic functional τ over $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$ is said to be *continuous* if for any chain of functions $\{f_i\}$

$$\tau(\text{lub}\{f_i\}) \equiv \text{lub}\{\tau(f_i)\}.$$

Note that since $\{f_i\}$ is a chain and τ is monotonic, $\tau(f_0) \prec \tau(f_1) \prec \tau(f_2) \prec \dots$; i.e., $\tau(\{f_i\})$ is also a chain. Therefore, by Lemma 5.2, both $\text{lub}\{f_i\}$ and $\text{lub}\{\tau(f_i)\}$ must exist.

We view τ as the composition operator of known monotonic functions, i.e., for $P_0 = S_0; P'_0$ with S_0 a statement in $L(G)$, and P_0, P'_0 program segments in $L(G)$, $P_0 = S_0; P'_0 \Rightarrow S_0; P'_0 \equiv \tau(P_0) = P'_0$. The following theorem shows that τ is continuous. It is a variant of a theorem by Manna.

Theorem 5.6 *The functional τ defined by composition of monotonic functions is continuous.*

Proof:

If τ consists of just S , a monotonic function, then τ is clearly continuous. We showed monotonicity of program statements by naturally extending them. We show continuity of τ for the **while \mathcal{G} do S od** statement. The proof consists of showing

- (i) $\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is monotonic and continuous for one application of the statement.
- (ii) $\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is monotonic and continuous for multiple applications of the statement.

Proof: of (i).

The proof is in two parts. First, $\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is monotonic, second, $\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is continuous. That is, you can operate on the least upper bound of a chain and get the same results as if you operate on the least upper bound of all of its elements, i.e., continuity preserves limits. Suppose **while \mathcal{G} do S od**; and $S = S1; S2; \dots; Sn$ is a chain of program statements.

$\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is monotonic.

If \mathcal{G} is initially false or \mathcal{G} is undefined, then **while \mathcal{G} do S od** is monotonic by natural extension, so $\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is monotonic. Suppose \mathcal{G} is true for one application. Then τ maps $\tau(S1; S2; \dots; Sn)$. We write this

$$\tau(\underbrace{\tau(\dots \tau(S)) \dots}_{n \text{ times}}) = \tau(\tau^n(S))$$

to denote that each Si , $1 \leq i \leq n$, is mapped to its logical successor by τ . We assume that τ^i , $1 \leq i \leq n$, is monotonic because the Si , $1 \leq i \leq n$, are not iterative, and show that τ^{i+1} is monotonic. First, if $Si \prec Si'$, $1 \leq i \leq n$, then by monotonicity of $\tau^1, \tau^2, \dots, \tau^n$ we have that $\tau^j(Si) \prec \tau^j(Si')$, $1 \leq j \leq n$. Then, since by natural extension **while \mathcal{G} do S od** is monotonic, $\tau^{i+1}(\tau^i(Si)) \prec \tau^{i+1}(\tau^i(Si'))$. So $\tau^{i+1}(Si)$ is

a monotonic functional.

$\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is continuous.

Now we show that τ is continuous, i.e., we show that $\tau(\text{lub}\{Si\}) \equiv \text{lub}\{\tau(Si)\}$ for any chain $\{Si\}$ within the scope of the $\text{while } \mathcal{G} \text{ do } S \text{ od}$ statement. Since $Si \prec \text{lub}\{Si\}$ for every $i \geq 0$ (each command approximates the lub of the chain of commands by the definition of least upper bound of a chain), by monotonicity of the functional τ^1, \dots, τ^n and the $\text{while } \mathcal{G} \text{ do } S \text{ od}$ statement, we have $\tau(Si) \prec \tau(\text{lub}\{Si\})$ for every $i \geq 0$. Therefore, $\text{lub}\{\tau(Si)\} \prec \tau(\text{lub}\{Si\})$.

Now let $\vec{x} \in (\mathcal{N}_\omega)^n$. Since the Si , $1 \leq i \leq n$, are not iterative, we assume that τ^i , $1 \leq i \leq n$, is continuous. We have defined $\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ as $\tau^{i+1}(\tau^i(\dots \tau(S))\dots)$ so:

$$\begin{aligned} \tau^{i+1}(\text{lub}\{Si\})(\vec{x}) &\equiv \tau^{i+1}(\tau^i(\dots (\tau(\text{lub}\{Si\})(\vec{x}))\dots)) \\ &\equiv \tau^{i+1}(\text{lub}\{\tau^i(\tau^{i-1}(\dots (\tau(Si)(\vec{x}))\dots))\}). \end{aligned}$$

From Lemma 5.2 we know that for every j , $1 \leq j \leq n$, there is an i_j such that for every $k \geq i_j$:

$$\text{lub}\{\tau^j(Si)\}(\vec{x}) \equiv \tau^j(Sk)(\vec{x}).$$

(every chain has a lub).

Let i_0 be the maximum of i_1, \dots, i_n (we have a series of statements Si). Then for every j , $1 \leq j \leq n$:

$$\text{lub}\{\tau^j(Si)\}(\vec{x}) \equiv \tau^j(Si_0)(\vec{x}),$$

and so

$$\begin{aligned} &\tau^{i+1}(\text{lub}\{\tau^i(\tau^{i-1}(\dots (\tau(Si)(\vec{x}))\dots))\}) \\ &\equiv \tau^{i+1}(\tau^i(\tau^{i-1}(\dots (\tau(Si)(\vec{x}))\dots))) \\ &\equiv \tau^{i+1}(\tau^i(\tau^{i-1}(\dots (\tau(Si))\dots)))(\vec{x}) \\ &\equiv \tau^{i+1}(Si_0)(\vec{x}) \\ &\prec \text{lub}\{\tau^{i+1}(Si)\}(\vec{x}) \end{aligned}$$

And since \vec{x} was an arbitrary element of the domain,

$$\tau^{i+1}(\text{lub}\{Si\}) \prec \text{lub}\{\tau^{i+1}(Si)\}.$$

so

$$\tau(\text{lub}\{Si\}) \prec \text{lub}\{\tau(Si)\}.$$

This concludes the proof of part (i). It shows that τ is continuous for all of our noniterative program statements and for one iteration of the **while** \mathcal{G} **do** S **od**. Next we show that τ is continuous for multiple iterations of the **while** \mathcal{G} **do** S **od**.

Proof: of (ii)

Let W denote **while** \mathcal{G} **do** S **od**. From the operational semantic definition of W in Chapter 4, we must show that for S a statement in $L(G)$, **while** \mathcal{G} **do** S **od**; \Rightarrow S ; **while** \mathcal{G} **do** S **od** $\equiv \tau(\text{while } \mathcal{G} \text{ do } S \text{ od}) = \text{while } \mathcal{G} \text{ do } S \text{ od}$. That is, τ correctly maps the last executable statement within a while loop to the first executable statement when the guard tests true.

There are two clarifying remarks we can make about this part of the proof:

1. We will presently define $\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ to be

$$\tau(\underbrace{\tau(\dots \tau(W)) \dots}_{n \text{ times}}) = \tau(\tau^n(W))$$

This is the case where we have termination. From here the proof is very similar to the previous proof, except that now we have monotonicity of each one of the $\tau^i(Wi)$, $1 \leq i \leq n$, and we use this to show that $\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is a monotonic functional for multiple iterations.

For continuity, when showing that

$$lub\{\tau(Wi)\} \prec \tau(lub\{Wi\}),$$

we recognize that $\{Wi\} \prec lub\{Wi\}$ instead of that $Si \prec lub\{Si\}$ as we did previously, because each occurrence of $\tau^i(Wi)$, $1 \leq i \leq n$, is now a chain.

2. Finally, we must show that the result holds in the general case when τ contains any number of occurrences of W .

The proof is in two parts. First, $\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is monotonic, second, $\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is continuous for multiple applications. Suppose $\text{while } \mathcal{G} \text{ do } S \text{ od} = W$, and \mathcal{G} is true for multiple applications.

$\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is monotonic.

Since \mathcal{G} is true for multiple applications, τ maps $\tau(W1; W2; \dots; Wn)$. We write this

$$\tau(\underbrace{\tau(\dots \tau(W)) \dots}_{n \text{ times}}) = \tau(\tau^n(W))$$

to denote that each Wi , $1 \leq i \leq n$, is mapped to its logical successor by τ . This assumes that the loop terminates. In the previous proof we showed that $\tau(W)$ is continuous for one application, therefore it is monotonic, since continuity ensures monotonicity. For each i , $1 \leq i \leq n$, Wi is continuous, so τ^i is monotonic for each i , $1 \leq i \leq n$. Now we show that $\tau^{i+1}(Wi)$ is monotonic. First, if $Wi \prec Wi'$, $1 \leq i \leq n$, then $\tau^j(Wi) \prec \tau^j(Wi')$, $1 \leq j \leq n$. By natural extension, the function $\text{while } \mathcal{G} \text{ do } S \text{ od}$ is monotonic, so $\tau^{i+1}(\tau^i(Wi)) \prec \tau^{i+1}(\tau^i(Wi'))$. So $\tau^{i+1}(Wi)$ is monotonic over n applications.

$\tau(\text{while } \mathcal{G} \text{ do } S \text{ od})$ is continuous over n applications.

That is, $\tau(lub\{Wi\}) \equiv lub\{\tau(Wi)\}$ for any chain $\{Wi\}$ (multiple applications of the $\text{while } \mathcal{G} \text{ do } S \text{ od}$ statement). Since $\{Wi\} \prec lub\{Wi\}$ for every $i \geq 0$, by monotonic-

ity of the functional τ^i , $1 \leq i \leq n$, and the **while** \mathcal{G} **do** S **od** statement (from the previous proof, they are continuous therefore monotonic), we have $\tau(Wi) \prec \tau(\text{lub}\{Wi\})$ for every $i \geq 0$. Therefore, $\text{lub}\{\tau(Wi)\} \prec \tau(\text{lub}\{Wi\})$.

Now let $\vec{x} \in (\mathcal{N}_\omega)^n$. We know τ^i , $1 \leq i \leq n$, is continuous (therefore monotonic), and since $\tau(Wi)$ is defined as $\tau^{i+1}(\tau^i(\dots \tau(W)))$:

$$\begin{aligned} \tau^{i+1}(\text{lub}\{Wi\})(\vec{x}) &\equiv \tau^{i+1}(\tau^i(\dots (\tau(\text{lub}\{Wi\})(\vec{x}))) \dots)) \\ &\equiv \tau^{i+1}(\text{lub}\{\tau^i(\tau^{i-1}(\dots (\tau((Wi)(\vec{x}))) \dots))\}). \end{aligned}$$

From Lemma 5.2 we know that for every j , $1 \leq j \leq n$, there is an i_j such that for every $k \geq i_j$:

$$\text{lub}\{\tau^j(Wi)\}(\vec{x}) \equiv \tau^j(Wk)(\vec{x}).$$

(every chain has a *lub*).

Let i_0 be the maximum of i_1, \dots, i_n (we have a series of statements Wi). Then for every j , $1 \leq j \leq n$:

$$\text{lub}\{\tau^j(Wi)\}(\vec{x}) \equiv \tau^j(Wi_0)(\vec{x}),$$

and so

$$\begin{aligned} &\tau^{i+1}(\text{lub}\{\tau^i(\tau^{i-1}(\dots (\tau((Wi)(\vec{x}))) \dots))\}) \\ &\equiv \tau^{i+1}(\tau^i(\tau^{i-1}(\dots (\tau(Wi)(\vec{x}))) \dots)) \\ &\equiv \tau^{i+1}(\tau^i(\tau^{i-1}(\dots (\tau(Wi)) \dots)))(\vec{x}) \\ &\equiv \tau^{i+1}(Wi_0)(\vec{x}) \\ &\prec \text{lub}\{\tau^{i+1}(Wi)\}(\vec{x}) \end{aligned}$$

And since \vec{x} was an arbitrary element of the domain,

$$\tau^{i+1}(\text{lub}\{Wi\}) \prec \text{lub}\{\tau^{i+1}(Wi)\}$$

so

$$\tau(\text{lub}\{Wi\}) \prec \text{lub}\{\tau(Wi)\}.$$

This shows that $\tau(W)$ is continuous for n applications, with n ranging over the natural numbers since n is arbitrary. \square

5.3.5 Fixed Points of the Functional τ

Fixed points of continuous functionals are states in a computational chain where the specific purpose or goal of the computation is met because they represent the best approximation to the limit of the computation [7]. Formally, let τ be a functional over $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$. We say that a function $f \in \{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$ is a *fixed point* of τ if $\tau(f) = f$. If f is a fixed point of τ and $f \prec g$ for any other fixed point g of τ , then f is called the *least fixed point* of τ .

τ is a continuous functional over $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$. Let $\tau^0(\omega) = \omega$ be the totally undefined function. Consider the sequence of functions $\tau^0(\omega), \tau^1(\omega), \tau^2(\omega), \dots$, where $\tau^{i+1}(\omega)$ is $\tau(\tau^i(\omega))$ for $i \geq 0$. Each function $\tau^i(\omega)$ is a member of $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$. By definition, $\omega \prec \tau(\omega)$, and since τ is monotonic (it is continuous, therefore monotonic), $\{\tau^i(\omega)\}$ must be a chain. That is:

$$\omega \prec \tau(\omega) \prec \tau^2(\omega) \dots$$

So by the least upper bound lemma, $\text{lub}\{\tau^i(\omega)\}$ must exist. That is, for the continuous functional τ over $\{(\mathcal{N}_\omega)^n \rightarrow_M (\mathcal{N}_\omega)^n\}$, the sequence $\{\tau^i(\omega)\}$ is a chain and has a least upper bound.

Because of this property, we can invoke Kleene's First Recursion Theorem.

Theorem 5.7 [First Recursion Theorem]

The continuous functional τ has a least fixed point that is the least upper bound of $\{\tau^i(\omega)\}$.

Denote the least upper bound of $\{\tau^i(\omega)\}$ by fp_τ . fp_τ is a fixed point of τ because τ

is continuous so

$$\tau(fp_\tau) = \tau(\text{lub}\{\tau^i(\omega)\}) = \text{lub}\{\tau^{i+1}(\omega)\} = \text{lub}\{\tau^i(\omega)\} = fp_\tau.$$

fp_τ is a least fixed point of τ as well, i.e., $fp_\tau \prec g$ for any fixed point g of τ . First of all, $\tau^0(\omega) = \omega \prec g$. Then, if $\tau^{i-1}(\omega) \prec g$ for some $i \geq 1$, since τ is monotonic (because τ is continuous) and g is a fixed point of τ , we have that $\tau^i(\omega) = \tau(\tau^{i-1}(\omega)) \prec \tau(g) = g$. So $\tau^i(\omega) \prec g$ for all $i \geq 0$. This implies that g is an upper bound of $\{\tau^i(\omega)\}$, but since fp_τ is the least upper bound of $\{\tau^i(\omega)\}$, it must be that $fp_\tau \prec g$.

We have defined operational syntax that constructs computation sequences using the functional “,” that we have proven is continuous for all of our program statements. This gives us the existence of a least upper bound for any sequence because we have defined our state space to be a complete lattice with \emptyset the least element. Note that $\emptyset \equiv FALSE \equiv \perp \equiv \# \equiv \omega$ are used to represent *undefined* at the various levels of abstraction in our environment. Thus, even for partial functions for which no value is defined, we have that $\text{lub}\{\} = FALSE$, and for partial functions defined on their input and for all total functions, $\text{lub}\{\Theta\} = \gamma \in \Gamma$ (where Θ represents a nonempty sequence), which by definition has a descriptive logical sentence.

CHAPTER 6

The *wpw* Predicate Transformer

The *wp* predicate transformer uses the initial state/final state pair to reason about correctness. In this model all program statements are atomic, even loops. While the semantics of the *wp* predicate transformer allude to execution of a loop, the proof of termination ignores it by focusing only on final results. The class of functions addressed by the *wp* method is the primitive recursive class. It is an easy exercise to show that all primitive recursive functions are computed by bounded loop programs [4],[22],[26].

Our interest is in verifying the correctness of unbounded loop programs. Whereas bounded loop programs require a given number of applications of a loop to its arguments and then terminate, unbounded loop programs correspond to repeated applications of a loop to its arguments until some condition of the arguments is satisfied. Unbounded loop programs are defined by each iteration of the loop, as Sommerhalder and van Westrhenen formalize [22].

Definition 6.1 *The i -iteration of a function f is a function from $(\mathcal{N}_\omega)^n$ to $(\mathcal{N}_\omega)^n$ denoted by f^{∇_i} and defined as follows.*

1. Let $\nabla_i(\vec{x}) \triangleq \{m \mid f^m(\vec{x}) \text{ is defined}\}$.

That is, $\nabla_i(\vec{x})$ is the set of indexes m such that $f(\vec{x})$ is defined for m iterations.

2. Let $\min \nabla_i(\vec{x})$ denote the least element of $\nabla_i(\vec{x})$. Then $f^{\nabla_i} \triangleq \lambda \vec{x} [\text{if } (\nabla_i(\vec{x}) \neq \emptyset \text{ and } f^m(\vec{x}) \downarrow \text{ for all } m \leq \min \nabla_i(\vec{x})) \text{ then } f^{\min \nabla_i(\vec{x})}(\vec{x}) \text{ else } \uparrow]$.

Note that the fixed point, $f^{\nabla_i}(\vec{x})$, is the least upper bound of a well ordered sequence of approximations that defines the loop.

In our model loops are not atomic because each iteration is represented as an element in the computational chain. Dijkstra applied his iterative function to its variables once and obtained a single predicate as his result, enabling him to define iteration as a recurrence relation between predicates. We apply the iterative function repeatedly, each time to updated values of the variables, and obtain a new predicate at each iteration. In essence, we iterate through a *family* of functions. Because of this we need a series of maps from predicate to predicate, so we define iteration as a recurrence relation between predicate transformers. At this point Dijkstra's statement about the difficulty of general recursion becomes clear: "...the semantics of a repetitive construct can be defined in terms of a recurrence relation between predicates, whereas the semantic definition of general recursion requires a recurrence relation between predicate transformers. This shows quite clearly why I regard general recursion as an order of magnitude more complicated than just repetition" [1].

6.1 Macros

Writing programs that use only subscripted elements of a program vector can be tedious. More importantly, because we wish to express the set of predicates that describe each element of a computational chain, we need notation to distinguish them. Like Hehner [8], we use the convention of inventing a name for a program segment, using the name in place of the segment, and specifying the text of the segment elsewhere. We use these names directly in our specifications to reason about the code segment the specification describes. Hehner referred to the use of a name

in place of some statement as a “call.” We use the term “macro.” In both cases the process is one of simple inline code substitution. Hehner used his call names as placeholders in a recursive refinement. We use our macro names directly in our specifications to reason logically about the correctness of the program segment that uses them.

Example 6.1 *The program $Q = (p, k, n, P) = (2, 1, 4, x_4 := x_3; x_1 := x_2; \text{while } x_4 \neq 0 \text{ do } x_1 := x_1 + 1; x_4 := x_4 - 1; \text{od}; \text{end.})$ adds the value of x_2 to the value of x_1 .*

We allow giving variables arbitrary names that are strings of characters beginning with a character. All of the registers used must be listed in a header that names the macro. The program variables naming the registers must be listed in a specific order: output variables first, next input variables, last local variables.

macro *ADD*(registers: *sum, x, y, count*);

count := *y*;

sum := *x*;

while *count* $\neq 0$ **do**

sum := *sum* + 1;

count := *count* - 1;

od;

To convert a macro into the program segment it represents, number the variables in the register list upwards from 1 in the order given. Remove the header with the register list.

To transform macro ADD into the code segment it represents:

1. *Number the variables in the register list: $sum = x_1, x = x_2, y = x_3, count = x_4$.*

2. *Remove the header with the register list.*

3. *The program sequence is:*

$x_4 := x_3; \quad \Leftrightarrow \quad count := y;$

$x_1 := x_2;$	\Leftrightarrow	$sum := x;$
while $x_4 \neq 0$ do	\Leftrightarrow	while $count \neq 0$ do
$x_1 := x_1 + 1;$	\Leftrightarrow	$sum := sum + 1;$
$x_4 := x_4 - 1;$	\Leftrightarrow	$count := count - 1;$
od;	\Leftrightarrow	od;

Each occurrence of a macro name in a program sequence must be converted to the program segment it represents before it is executed. To make use of a macro name in a program we use a macro statement: $x := ADD(x, y)$.

6.1.1 Syntax of Macros

Macros involve simple in-line substitution of program text. Macros may be composed of any legal statement in our language. The syntax of a macro is:

macro< *string* > (**registers** : < *string*₁ >, < *string*₂ >, ..., < *string*_n >);

 < *sequence* >;

< *string* >::= < *character* > | < *string* > < *character* > | < *string* > < *numeral* >

< *character* >::= A|B|...|Z|a|b|...|z

The macro statements are:

- $(z_1, \dots, z_k) := f(x_1, \dots, x_p),$
- $(z_1, \dots, z_k) := (z_1, \dots, z_k) + f(x_1, \dots, x_p),$
- $(z_1, \dots, z_k) := (z_1, \dots, z_k) - f(x_1, \dots, x_p),$

where $f : \mathbb{N}^p \rightarrow \mathbb{N}^k$ is a computable function. The above macro statements stand for a sequence P computing the function value $f(x_1, \dots, x_p) = (y_1, \dots, y_k)$ and assigning

the values of the variables y_i to the variables z_i , $1 \leq i \leq k$. The values of all other variables occurring in the program containing the macro statement remain unchanged.

6.1.2 Use of Macro Names

We use macro names directly in our specifications. Macros allow us to name specific code sequences.

Example 6.2 *Suppose we have this code sequence:*

$S_1;$
 $S_2;$
 $:$
 $S_n;$

We can name the sequence $S_1; S_2; \dots; S_n$ “S” and we can verify the correctness of S by verifying the correctness of $S_1; S_2; \dots; S_n$.

A transformation of a macro name to the code it represents consists of a direct substitution of the code sequence for the macro name in such a way that side effects are not allowed. A macro names the partial function computed by the code sequence the macro represents. We reason about the correctness of a computation of the partial function with the preconditions and postconditions we write for the code sequence that computes it. Because a macro represents this code sequence, the preconditions and postconditions of the code sequence are preconditions and postconditions for the macro.

When we use a macro to name a partial function in a computational sequence, we may interpret the function name as a predicate.

Example 6.3 *Consider a representation of a computational chain of a simple code segment, with precondition Q , invariant I , and postconditions R_i , $1 \leq i \leq 5$. This small program, similar to Example 6.1, is composed of two sequential statements (S_1*

and $S2$), followed by a loop containing statements $S3$ and $S4$ that executes twice. The postcondition of the loop is $R5$.

$$\{Q\}S1; \{R1\}S2; \{R2\}\{I\}S3; \{R3\}S4; \{R4\}\{I\}S3; \{R3\}S4; \{R4\}\{R5\}$$

A macro will allow us to name a piece of code, say $P = S3; S4$ and verify its correctness once. When we verify the entire code segment, we can substitute this name in a computational chain. For verification of the entire code segment the computational chain becomes

$$\{Q\}S1; \{R1\}S2; \{R2\}\{I\}P; \{I\}P; \{R5\}$$

We can write a postcondition asserting the correctness of the macro since we have verified it elsewhere, say $R5'$ (because each execution of P should provide a better approximation to the state where $R5$ is fulfilled), and the computational chain becomes

$$\{Q\}S1; \{R1\}S2; \{R2\}\{I\}P; \{R5'\}\{I\}P; \{R5'\}\{R5\}$$

Since our computational chains are well-ordered we have a structure we can investigate inductively. The result is a logical and consistent method for reasoning about bounded, unbounded, and nonterminating computations.

6.2 The *wpw* Predicate Transformer

We denote our predicate transformer *wpw*. Like the *wp* predicate transformer our *wpw* predicate transformer is a conjunction. The *wp* predicate transformer is a conjunction of termination and partial correctness [1], [19]:

$$wp(A, R) \equiv wp(A, TRUE) \wedge wlp(A, R).$$

$wlp(A, R)$ states that the computation will be correct if it terminates. $wp(A, TRUE)$ guarantees that the computation will terminate after a pre-defined number of iterations of any loop [1].

Our wpw predicate transformer includes the wlp predicate transformer which takes a command and its postcondition as arguments because the wlp predicate transformer asserts that the postcondition will be fulfilled if the command terminates.

The other conjunct of the wpw predicate transformer expresses incremental correctness. This conjunct defines a recurrence relation on an ascending chain of predicate transformers. For noniterative commands, the chains are composed of a single element and the arguments to the predicate transformer of that element are the command and its postcondition. For the iterative command the chains may be finite or infinite. The arguments to the predicate transformers in the chains of predicate transformers are the command obtained by combining the statements of the guarded command and the logical assertion of the command's specification. We interpret this assertion as a logical approximation to the predicate describing the limit of the computation for each iteration. Each appearance of this assertion in a computational chain represents the least upper bound of a subchain within the chain, and describes the most recent member of the family of functions to execute. We describe this assertion by appending a prime (\prime) to the letter we use to represent the postcondition (see Section 2.4).

We adopt the same semantic definition for statements where termination is not an issue as Dijkstra, except that the semantic definition of our decision construct reflects its deterministic nature. The semantic equations for our noniterative commands are:

1. Assignment: $wpw("x := e", R) = R_e^x$
2. Sequencing: $wpw("S1; S2", R) = wpw("S1", wpw("S2", R))$
3. Decision: $wpw("if \mathcal{G} then S1 else S2 fi", R) = \mathcal{G} \Rightarrow wpw("S1", R) \wedge \neg \mathcal{G} \Rightarrow$

$wpw("S2", R)$

4. $wpw("end.", R) = R$ since $end.$, like Dijkstra and Gries "*skip*" is a no-op.

6.2.1 Formal Definition of an Unbounded Loop

Let *WHILEDO* represent the command while \mathcal{G} do S od. Let *IF* represent the command if \mathcal{G} then S fi. The formal semantic definition of our looping command is given by cases using the predicate $I_k(R)$ which describes the set of all states from which *WHILEDO* can execute k times leaving R true.

Definition 6.2 For guard \mathcal{G} , postcondition R , and approximation R' :

- $I_0(R) = FALSE$. This predicate represents the set of states before the guard is initially tested.
- $I_{k+1}(R) = I_k(R) \vee wpw(IF, I_k(R'))$, $k > 0$. This predicate represents the set of all states from which *WHILEDO* has executed correctly k times such that either $\neg\mathcal{G} \wedge R' \wedge R$ after k iterations or $\mathcal{G} \wedge R' \wedge \neg R$, i.e. either the loop has terminated and R is true ($I_k(R)$) or another iteration is required ($wp(IF, I_k(R'))$).

Note the similarity between this notation and Dijkstra's notation [1]. The principal difference is that our chains are ascending, while Dijkstra begins with a descending chain that he proves is bounded from below by zero, since he always assumes termination *a priori*. Given this definition we can define our semantic operator for our iterative command.

Definition 6.3 $wpw(WHILEDO, R) = (\forall k : k > 0 : I_k(R))$.

This definition says that either termination and $I_{k-1}(R)$ is true or the computation is correct to this point and one more iteration is needed $wp(IF, I_{k-1}(R'))$.

6.2.2 Incremental Progress

In the definition above we gave the formal definition of a loop in terms of the chain of predicates that describe the computation. Next we propose a definition for a loop in terms of the predicate transformers we use to evaluate the chain of predicates. We will show that this proposed definition exactly defines the semantics of a loop in our model.

Definition 6.4 Incremental Progress

$$wpw(WHILEDO, R) \equiv wp_i(S, R'_i) \wedge wlp(WHILEDO, R), i = 0, 1, 2, \dots$$

where $wp_i(S, R'_i) \equiv wp(S, TRUE) \wedge wlp(S, R'_i)$.

This predicate transformer captures the meaning of our formal definition because it addresses the situation where termination has occurred and R is realized ($wlp(WHILEDO, R)$) [1],[19],[6]. It also addresses the situation where more iteration is required ($wp_i(S, R'_i)$) but the computation is correct to this point. In this case the statements within the loop have executed i times, and for each discrete execution j , $1 \leq j \leq i$, we have reached a state that represents the least upper bound of the computational subchain induced by those statements. Each of these least upper bounds has a predicate logic sentence identified by the predicate R'_j .

Continuity of our computational chains ensures we can obtain better and better approximations to a computation's limit as we map from predicate transformer to predicate transformer. We will show that at each fixed point the command of our predicate transformer is more defined than its predecessor because its nucleus is larger. Monotonicity ensures that $wp_1(S, R'_1) \Rightarrow \dots \Rightarrow wp_{i-1}(S, R'_{i-1}) \Rightarrow wp_i(S, R'_i) \Rightarrow \dots$.

Theorem 6.1 Incremental Progress Theorem

$$wpw(WHILEDO, R) \equiv wp_i(S, R'_i) \wedge wlp(WHILEDO, R), \quad i = 0, 1, 2, \dots$$

where $wp_i(S, R'_i) \equiv wp(S, TRUE) \wedge wlp(S, R'_i)$.

Proof: Induction on i , the index of a state in a computational chain.

- **Basis:** $i = 0$. This is the initial state before execution, described by \emptyset , the minimal element in our lattice. Since S has not executed, $R'_0 = \omega$, so

$$\begin{aligned}
 wpw(WHILEDO, R) &= wp_0(S, \omega) \wedge wlp(WHILEDO, R) \\
 &= \omega \wedge wlp(WHILEDO, R) \\
 &= \{\} \\
 &= FALSE
 \end{aligned}$$
- **Hypothesis:** We assume that for each index $i > 0$ in the chain, we have that S executes once, i.e., $wp(S, TRUE)$ and for that execution we know that $wlp(S, R'_i)$. Thus, because of the structure of our state space and by continuity of computational chains, we have $N(S^i) = (S^i)(S^i)^{-1} = wp_i(S, R'_i)$, i.e., our initial states were such that after i iterations R'_i is true, and iterated composition of S with itself produces a nucleus of states that map into the *lub* of the chain, the state described by R'_i .
 - **Case A:** Index i is the *lub* of the entire computation, i.e., the loop has terminated. In this case we have $Invariant \wedge R'_i \wedge \neg \mathcal{G} \Rightarrow R$, since R'_i is met and the loop has terminated we certainly have $wlp(WHILEDO, R)$, so $wpw(WHILEDO, R) \equiv wp_k(S, R'_k) \wedge wlp(WHILEDO, R), 0 \leq k \leq i$.
 - **Case B:** Index i is the *lub* of a subsequence of the entire computational chain, i.e., the loop has not terminated and R'_i is an approximation to R . $wlp(WHILEDO, R)$ insures partial correctness of the loop;

it is the set of initial states from which R will be satisfied if the computation terminates. We have already shown by continuity of computational chains that $wp_k(S, R'_k), 0 \leq k \leq i$, therefore to this point we have $wpw(WHILEDO, R) \equiv wp_k(S, R'_k) \wedge wlp(WHILEDO, R), 0 \leq k \leq i$.

- **Show:** $wpw(WHILEDO, R) \equiv wp_{i+1}(S, R'_{i+1}) \wedge wlp(WHILEDO, R)$. By continuity of computational chains one more iteration of the loop produces the result we need, because by iterated composition of the command S with itself we have $N(S^{i+1}) = (S^i S)(S^i S)^{-1} = wp_{i+1}(S, R'_{i+1})$. The two cases, either $i + 1$ is a final state or $i + 1$ is a non-final state, are the same as we assumed in our hypothesis, with $i + 1$ replacing i . So $wpw(WHILEDO, R) \equiv wp_{i+1}(S, R'_{i+1}) \wedge wlp(WHILEDO, R), i = 0, 1, 2, \dots$.

6.2.3 Properties of the wpw Predicate Transformer

Like Nelson [6] and de Bakker [5] we eliminate Dijkstra's *Law of the Excluded Miracle*: $wp(S, FALSE) = FALSE$ because nonterminating computations return an empty set of final states. There are other properties that are important to our reliance on computational chains for verifying the correctness of programs.

- **Property 1a:** The predicate transformer wpw is monotonic for terminating loops. Let $(Q' \wedge Q) \Rightarrow (R' \wedge R)$. Then

$$wpw(S, (Q' \wedge Q)) \Rightarrow wpw(S, (R' \wedge R)).$$

Proof: $(Q' \wedge Q) \wedge (R' \wedge R) = (Q' \wedge Q)$. Therefore

$$\begin{aligned} wpw(S, (Q' \wedge Q)) &= wpw(S, (Q' \wedge Q) \wedge (R' \wedge R)) \\ &= wpw(S, (Q' \wedge Q)) \wedge wpw(S, (R' \wedge R)) \\ &\Rightarrow wpw(S, (R' \wedge R)). \end{aligned}$$

- **Property 1b:** The predicate transformer wpw is monotonic for nonterminating loops. For iterations i and j , with $i < j$, let $Q'_i \Rightarrow Q'_j$. Then

$$wpw(S, Q'_i) \Rightarrow wpw(S, Q'_j).$$

Proof: $Q'_i \wedge Q'_j = Q'_i$. Therefore

$$\begin{aligned} wpw(S, Q'_i) &= wpw(S, Q'_i \wedge Q'_j) \\ &= wpw(S, Q'_i) \wedge wpw(S, Q'_j) \\ &\Rightarrow wpw(S, Q'_j). \end{aligned}$$

- **Property 2a:** The wpw predicate transformer is continuous for terminating loops.

$$wpw(S, Q' \wedge Q) \vee wpw(S, R' \wedge R) = wpw(S, (Q' \wedge Q) \vee (R' \wedge R)).$$

Proof:

(\Rightarrow): Let $\gamma \in wpw(S, Q' \wedge Q)$ or let $\gamma \in wpw(S, R' \wedge R)$. Then $\gamma \in wpw(S, (Q' \wedge Q) \vee (R' \wedge R))$.

(\Leftarrow): Let $\gamma \in wpw(S, (Q' \wedge Q) \vee (R' \wedge R))$. Then either $\gamma \in wpw(S, Q' \wedge Q)$ or $\gamma \in wpw(S, R' \wedge R)$, or both. If both, then we are done. If $\gamma \in wpw(S, Q' \wedge Q)$ then $\gamma \in wpw(S, (Q' \wedge Q) \vee wpw(R' \wedge R))$. If $\gamma \in wpw(S, R' \wedge R)$ then $\gamma \in wpw(S, (Q' \wedge Q) \vee (R' \wedge R))$.

- **Property 2b:** The wpw predicate transformer is continuous for nonterminating loops. For iterations i and j , with $i < j$:

$$wpw(S, Q'_i) \vee wpw(S, Q'_j) = wpw(S, Q'_i \vee Q'_j).$$

Proof:

(\Rightarrow): Let $\gamma \in wpw(S, Q'_i)$ or let $\gamma \in wpw(S, Q'_j)$. Then $\gamma \in wpw(S, Q'_i \vee Q'_j)$.

(\Leftarrow): Let $\gamma \in wpw(S, Q'_i \vee Q'_j)$. Then either $\gamma \in wpw(S, Q'_i)$ or $\gamma \in wpw(S, Q'_j)$, or both. If γ is in both, then we are done. If $\gamma \in wpw(S, Q'_i)$ then $\gamma \in wpw(S, Q'_i) \vee wpw(S, Q'_j)$. If $\gamma \in wpw(S, Q'_j)$ then $\gamma \in wpw(S, Q'_i) \vee wpw(S, Q'_j)$.

- **Property 3a:** The wpw predicate transformer distributes over conjunction in a terminating computation.

$$wpw(S, Q' \wedge Q) \wedge wpw(S, R' \wedge R) = wpw(S, (Q' \wedge Q) \wedge (R' \wedge R)).$$

Proof:

(\Rightarrow): Let $\gamma \in wpw(S, Q' \wedge Q)$ and let $\gamma \in wpw(S, R' \wedge R)$. Then $\gamma \in wpw(S, (Q' \wedge Q) \wedge (R' \wedge R))$.

(\Leftarrow): Let $\gamma \in wpw(S, (Q' \wedge Q) \wedge (R' \wedge R))$. Then $\gamma \in wpw(S, Q' \wedge Q)$ and $\gamma \in wpw(S, R' \wedge R)$. Therefore, $\gamma \in wpw(S, Q' \wedge Q) \wedge wpw(S, R' \wedge R)$.

- **Property 3b:** The wpw predicate transformer distributes over conjunction in nonterminating computations, provided that the index range of the predicates it takes as arguments is not empty. This restriction ensures we cannot map an undefined intermediate state to a defined state. For iterations i and j , with $i < j$:

$$wpw(S, Q'_i) \wedge wpw(S, Q'_j) = wpw(S, Q'_i \wedge Q'_j).$$

Proof:

(\Rightarrow): Let $\gamma \in wpw(S, Q'_i)$ and let $\gamma \in wpw(S, Q'_j)$. Then $\gamma \in wpw(S, Q'_i \wedge Q'_j)$.

(\Leftarrow): Let $\gamma \in wpw(S, Q'_i \wedge Q'_j)$. Then $\gamma \in wpw(S, Q'_i)$ and $\gamma \in wpw(S, Q'_j)$.

Therefore, $\gamma \in wpw(S, Q'_i) \wedge wpw(S, Q'_j)$.

6.2.4 Relationship Between I , R' , and R .

The logical relation between the invariant (I), the approximating predicate (R'), and the postcondition (R) of a loop is implication: $I \Rightarrow R'_1 \Rightarrow R'_2 \Rightarrow \dots \Rightarrow R$ whenever a loop is terminating.

If the loop is bounded then the invariant and the approximation may be the same. This work is not intended to address this case because it has been fully explored by others, and, given that we have *a priori* knowledge of the moment of termination, an approximation to the postcondition is superfluous. We have the fact in advance that the postcondition is the loop's specification and that it will be fulfilled. This research is valid for this case because R' can certainly be defined, but is intended to be used beyond the primitive recursive class of functions.

If a loop is unbounded but terminating (it computes a total recursive function), it is “controlled” by a Boolean guard that is a logical sentence describing the condition of the loop's variables (usually the condition we expect them to be in at termination). It may be the case here that several iterations are required before anything nontrivial can be said about the loop's variables. The example we give in Chapter 7 of a total recursive function is a good example of this phenomenon. Before the loop iterates at all the only variables it manipulates that we can comment on are the variables set elsewhere in the program. But one pass of the loop allows us to say much more because the loop has been able to compute a partial solution and set its variables to reflect this.

Where the invariant of a loop defines upper and lower bounds for some variables in the loop, the approximating predicate allows us to be more specific. First, it enables us to pinpoint the values within the bounds set by the invariant (for those variables the invariant can describe). Second it allows us to make a statement after each iteration about the variables involved with the guard of the loop (that the invariant may not

apply to). Third, it allows us to make a statement about the variables specific to the computation but not involved with the guard or the invariant. We do this by showing that each pass leaves $I \wedge R' = TRUE$.

If the loop is nonterminating then its postcondition is *FALSE*. The approximation predicate, true after each iteration, allows us to actually say something about the variables (Dijkstra could say nothing, Nelson could say *FALSE*, which is not very useful). In this case the relationship between the invariant and the approximation is still implication. The postcondition is not involved because it will never be reached. Technically, it doesn't exist. We have: $I \Rightarrow R'_1 \Rightarrow R'_2 \Rightarrow \dots$

CHAPTER 7

Application of the *wpw* Predicate Transformer

7.1 Description of the Method

We use the Incremental Progress Theorem to help us develop programs that compute recursive functions. The *wp* calculus method of predicate transformers is the motivation for our method and, while we have extended the range of programs for which proofs of correctness can be written using our method beyond what the *wp* calculus can do, we have attempted to design a method that is easy to use.

The concept of an invariant is fundamental to the correctness of all loops, bounded and unbounded. We write $\text{Invariant} \wedge \mathcal{G} \Rightarrow \text{wpw}(IF, \text{Invariant})$ to mean that one execution of the guarded command list S encompassed by **while** \mathcal{G} **do** S **od**, which is equivalent to executing **if** \mathcal{G} **then** S **fi** one time will not affect the truth of the invariant, provided it is true before execution commences. Given this, any number of iterations leaves the invariant true. Furthermore, since the invariant is independent of the guard of a loop it remains true if the loop terminates.

Our proof of correctness method addresses the truth of the loop invariant, both before execution and after each increment. Our proof of correctness method also

recognizes the important role of the specification of the code that comprises the loop body (the approximating predicate). Since it is the definition of the purpose of the code within the loop it must be fulfilled after the first iteration and remain true for all subsequent iterations. Finally our method recognizes that in conjunction with the invariant of the loop and a false guard the specification of the loop body implies the truth of the postcondition. Unfortunately, in this class of functions termination cannot always be predicted and so in the face of nontermination, we depend on the truth of the conjunction of the invariant and the specification of the loop body at each fixed point in the computational subchain to verify that our code can execute correctly as written.

To prove correctness using our method it is necessary to show the truth of the entites we have been discussing for every loop. We provide a checklist so that a program developer can be certain that nothing has been forgotten.

Checklist for showing correctness of a loop:

1. Show that the invariant (I) of the loop is true before execution of the loop begins.
2. Show that an execution of the loop body terminates with the invariant of the loop and the specification of the code within the loop both true. For postcondition R and the approximation to R , R' , this means show that $I \wedge R' = TRUE$. *R' is an approximation to the postcondition R . It is obtained by combining the postconditions of each statement within the loop by conjunction, and then simplifying the predicate logic sentence that results.*
3. Show that if termination occurs the postcondition is true, i.e., show that $I \wedge R' \wedge \neg \mathcal{G} \Rightarrow R$.

4. For S (the statement list within the scope of the guard) show $wp_i(S, R'_i), i = 1, 2, \dots$

7.2 Examples

Code for the programs we use may be found in Section 7.3 at the end of this chapter.

7.2.1 Proof of Correctness of a Total Recursive Function

We demonstrate our method on a total recursive (but not primitive recursive) function. The program that computes this function requires an unbounded loop, so even though it terminates, the methods of Dijkstra and Gries cannot be used to prove that it is correct.

We write $g^m(x)$ for the result of composition of g with itself m times. In particular,

$$g^0(x) = x,$$

$$g^1(x) = g(x),$$

$$g^2(x) = g(g(x)),$$

and so on. In general we can write this as the recursion

$$g^0(x) = x,$$

$$g^{m+1}(x) = g(g^m(x)).$$

For the function we want to compute we define

$$f_0(x) = \begin{cases} x + 1 & \text{if } x = 0 \text{ or } x = 1 \\ x + 2 & \text{otherwise,} \end{cases}$$

$$f_{n+1}(x) = f_n^x(1). \quad \text{Where } f_{n+1}(x) = f_n^x(1) = \underbrace{f_n(f_n \cdots (f_n(1)) \cdots)}_{x \text{ times}}$$

A basic property of this function is given by a lemma found in Davis and Weyuker [26].

Lemma 7.1 $f_{n+1}(x+1) = f_n(f_{n+1}(x))$.

Proof:

$$\begin{aligned} f_{n+1}(x+1) &= f_n^{x+1}(1) \\ &= f_n(f_n^x(1)) \\ &= f_n(f_{n+1}(x)) \end{aligned}$$

With this lemma we can show that $f_n(x) = A(n, x)$:

$$f_{i+1}(x+1) = f_i(f_{i+1}(x));$$

and

$$A(i+1, x+1) = A(i, A(i+1, x)).$$

For the base cases we have

$$\begin{aligned} A(i, 0) &= 1 = f_i(0) \\ A(0, x) &= \begin{cases} x+1 & \text{if } x = 0, 1 \\ x+2 & \text{if } x > 1. \end{cases} = f_0(x) \end{aligned}$$

The recursively specified function $A(i, x)$ is a variant of a function introduced by W. Ackermann.

We note that $f_n(0) = 1, n \geq 0$ and $f_n(1) = 2, n \geq 0$ since $f_n(1) = f_{n-1}(1) = \dots = f_0(1) = 2$. We can prove that this function is total by showing that $f_{n+1}(x) = f_n(f_n \dots (f_n(1)) \dots) = f_n^x(1)$, for all $n \geq 1$ and $x \geq 0$, i.e., the function is defined on all of its arguments. We demonstrate that it is computable by giving an algorithm that computes it and showing the proof of correctness using our method. We demonstrate that this function is not primitive recursive by the fact that it contains an unbounded loop. We must use an unbounded loop because the function f_i is monotonically increasing for all $i \geq 1$, i.e., we cannot find an *a priori* upper bound for the function in general because the runtime is dependent on the index of the function for all indices

greater than or equal to one.

We present our algorithm in the form of a macro to make it more easily readable. See Figure 7.1 in Section 7.3. The program calls one small macro that represents the base case of our functional specification, see Figure 7.3. The program calls the macro that computes values other than the base (see Figure 7.2) from within a bounded loop.

We prove the correctness of macro *FM1*. In this proof we show how our method works for bounded and unbounded loops, assignment statements, sequencing, and decision statements. Once the correctness of macro *FM1* is shown, the proof of the bounded loop in macro *Example1* in Section 7.1 is an easy exercise of the notation. The sequence operators \parallel and \nparallel have been defined in Chapter 4, Section 4.2.1 and Section 4.2.2. We use them here to manipulate a register we call *Stack*.

Proof of the bounded loop:

```

     $j := 0; i := i - 1; x := \nparallel Stack; x := x - 1;$ 
     $\{Pre : j = 0 \wedge pushing > 0\}$ 
     $\{I : 0 \leq j \leq pushing \wedge 0 \leq x \leq m - 1\}$ 
     $\{Bound : t = pushing - j\}$ 
    while  $j < pushing$  do
         $Stack := x \parallel Stack;$ 
         $i := i + 1;$ 
         $j := j + 1;$ 
         $\{R' : 1 \leq j \leq pushing \wedge Stack_i = x : i - j \leq i \leq i + j\}$ 
    od
     $\{R : j = pushing \wedge Stack_i = x : i - pushing \leq i \leq i + pushing\}$ 

```

1. Show that the invariant *I* is true before execution of the loop.

- $wpw("j := 0", 0 \leq j \leq \text{pushing})$
- $0 \leq 0 \leq \text{pushing}$
- $TRUE$

2. Show that an execution of the guarded command terminates with $I \wedge R'$ true.

- $wpw("Stack := x \parallel Stack; i := i + 1; j := j + 1", 0 \leq j \leq \text{pushing} \wedge 1 \leq j \leq \text{pushing} \wedge Stack_i = x : i - j \leq i \leq i + j)$
- $wpw("Stack := x \parallel Stack; i := i + 1", wpw("j := j + 1", 0 \leq j \leq \text{pushing} \wedge Stack_i = x : i - j \leq i \leq i + j))$
- $wpw("Stack := x \parallel Stack", wpw("i := i + 1", 0 \leq j + 1 \leq \text{pushing} \wedge (Stack_i = x : i - j + 1 \leq i \leq i + j + 1) = \omega))$
- $wpw("Stack := x \parallel Stack", 0 \leq j + 1 \leq \text{pushing} \wedge (Stack_{i+1} = x : i + 1 - j + 1 \leq i + 1 \leq i + 1 + j + 1) = \omega)$
- $0 \leq j \leq \text{pushing} \wedge Stack_i = x : i - j \leq i \leq i + j$. Implied by the invariant.

3. Show that upon termination the postcondition is true, i.e., show that $I \wedge R' \wedge \neg(\mathcal{G}) \Rightarrow R$.

- $0 \leq j \leq \text{pushing} \wedge 1 \leq j \leq \text{pushing} \wedge Stack_i = x : i - j \leq i \leq i + j \wedge \neg(j < \text{pushing}) \Rightarrow j = \text{pushing} \wedge Stack_i = x : i - \text{pushing} \leq i \leq i + \text{pushing}$.
- $j = \text{pushing} \wedge Stack_i = x : i - \text{pushing} \leq i \leq i + \text{pushing}$.

4. Show $wp_i(A, R'_i), i = 1, 2, \dots$

- **Basis:** If the index of the computational chain is 1 then we have executed once and $j = 1$ and by 2 we have that $0 \leq j \leq \text{pushing} \wedge Stack_i = x : i - j \leq i \leq i + j$. So $1 \leq j \leq \text{pushing} \wedge Stack_i = x : i - j \leq i \leq i + j$.
- **Hypothesis:** Assume for index $k > 1$ that $1 \leq j \leq \text{pushing} \wedge Stack_i = x : i - j \leq i \leq i + j$.

- **Show:** For index $k + 1$.

Case A: $j = \text{pushing}$. Then termination has occurred and by 3 we have that $I \wedge R' \wedge \neg(\mathcal{G}) \Rightarrow R$, i.e., $j = \text{pushing} \wedge \text{Stack}_i = x : i - \text{pushing} \leq i \leq i + \text{pushing}$.

Case B: $j < \text{pushing}$. Then the computation has advanced one more step and by 2 we have $0 \leq j + 1 \leq \text{pushing} \wedge \text{Stack}_{i+1} = x : i - j + 2 \leq i + 1 \leq i + j + 2$, i.e., $0 \leq j \leq \text{pushing} \wedge \text{Stack}_{i+1} = x : i + 1 - j \leq i + 1 \leq i + 1 + j$.

Proof of the major $IF \dots FI$ statement:

if $top \geq 1$ then

⋮

$\{j = \text{pushing} \wedge \text{Stack}_i = x : i - \text{pushing} \leq i \leq i + \text{pushing}\}$

else

⋮

$val := val + F0(1);$

fi

$\{(j = \text{pushing} \wedge \text{Stack}_i = x : i - \text{pushing} \leq i \leq i + \text{pushing}) \vee (top = 0 \wedge val = val + F0(1))\}$

Proof:

$wp(IF \dots FI, R') = (top \geq 1 \vee top = 0) \wedge$

- $(top \geq 1 \Rightarrow wp(\text{"IF} \dots FI", j = \text{pushing} \wedge \text{Stack}_i = x : i - \text{pushing} \leq i \leq i + \text{pushing}) \wedge$
- $(top = 0 \Rightarrow wp(\text{"val} := val + F0(1)", val = val + F0(1)))$
- $TRUE \wedge (top \geq 1 \Rightarrow j = \text{pushing} \wedge \text{Stack}_i = x : i - \text{pushing} \leq i \leq i + \text{pushing})$
- $\wedge (top = 0 \Rightarrow val = val + F0(1))$
- $TRUE \wedge TRUE \wedge TRUE$.

Proof of the unbounded loop:

```

i := 1; Stack := index || Stack; x := index;
{Pre : Stack1 = m − 1 ∧ val ≥ 0}
{I : i ≥ 0 ∧ 0 ≤ x ≤ m − 1}
{Bound : none}
while i ≠ 0 do
  if top ≥ 1 then
    :
  else
    :
  fi
  {(j = pushing ∧ Stacki = x : i − pushing ≤ i ≤ i + pushing) ∨ (top = 0 ∧ val =
    val + F0(1))}
  i := i − 1; top := ⌊ Stack;
  {R' : (top = 0 ∧ val = k * F0(1) : 1 ≤ k ≤ pushing) ∨ (top ≥ 1 ∧ Stacki = x :
    i − pushing ≤ i ≤ i + pushing)}
od
{R : i = 0 ∧ val = pushing * F0(1)} and pushing * F0(1) = F0pushing(1)

```

1. Show that the invariant *I* is true before execution of the loop.

- $0 \leq x \leq m - 1$ is true by the Precondition of the loop, since the stack only holds the value $m - 1$.
- $TRUE$.

2. Show that an execution of the guarded command terminates with $I \wedge R'$ true.

- $wpw(\text{"IF ... FI"; } i := i - 1, I \wedge R')$

- $= wpw("IF \dots FI", wpw("i := i - 1", I \wedge R'))$
- $= wpw("IF \dots FI", I \wedge R')$
- $= (top \geq 1 \wedge j = pushing \wedge Stack_i = x : i - pushing \leq i \leq i + pushing) \vee (top = 0 \wedge val = k * F0(1) : 1 \leq k \leq pushing)\}$ which is true from the proof of the if...fi statement, the truth of the invariant, and continuity of computational chains.

3. Show that upon termination the postcondition is true, i.e. show that $I \wedge R' \wedge \neg(\mathcal{G}) \Rightarrow R$.

- $i \geq 0 \wedge 0 \leq x \leq m - 1 \wedge (top \geq 1 \wedge Stack_i = x : i - pushing \leq i \leq i + pushing \vee top = 0 \wedge val = k * F0(1) : 1 \leq k \leq pushing) \wedge \neg(i \neq 0) \Rightarrow i = 0 \wedge val = k * F0(1)$.
- $0 \leq x \leq m - 1 \wedge i = 0 \wedge val = k * F0(1) \wedge 1 \leq k \leq pushing \Rightarrow i = 0 \wedge k = pushing \wedge val = pushing * F0(1)$, because the stack is empty, there were *pushing* copies of index 0 on the stack, and the stack is decremented at the end of each pass of the loop, so the line " $val := val + F0(1)$ " was executed exactly *pushing* times.

4. Show $wp_i(A, R'_i), i = 1, 2, \dots$

- **Basis:** The index in the computational chain is 1. Then by 2 we have that $i \geq 0 \wedge 0 \leq x \leq m - 1 \wedge R'$.
- **Hypothesis:** Assume that for index $k > 1$ we have $i \geq 0 \wedge 0 \leq x \leq m - 1 \wedge R'$.
- **Show:** For index $k + 1$.

Case A: $i = 0$. Then the stack is empty and termination has occurred. So by 3 we have $I \wedge R' \wedge \neg(\mathcal{G}) \Rightarrow R$, i.e., $i = 0 \wedge val = pushing * F0(1)$. But $val = pushing * F0(1) = F0^{pushing}(1)$, but this

is $f_{m-1}(f_{m-2}(\cdots(f_0(1))\cdots)) = f_{m-1}(1)$, because each time we iterate through the loop, the variable representing the index (x) is decremented.

- **Case B:** $i \neq 0$. Then the computation has advanced one more step and by 2 we have $I \wedge R'$.

Using the foundation of Dijkstra, neither Dijkstra nor Gries are able to say anything about the correctness of this program because of the *a priori* demonstration of termination that is required. Using our foundation we can show that the invariant we write for a loop is a true invariant. We can show that the conjunction of the loop's invariant and the approximation to the postcondition imply that each increment of the loop is correct. We can show that the conjunction of a loop's invariant, the approximation to the postcondition of the loop, and a negative guard for the loop implies that the postcondition is true. We can apply induction to each index of the computational chain induced by the program statements, and show that as the computation progresses each element of the computational chain has a logical description that either accurately approximates the limit of the computation or is itself the limit of the computation, depending on the condition of the loop's guard. With our method we can show the total correctness (partial correctness and termination) of this total recursive function.

7.2.2 Proof of Correctness of a Partial Recursive Function

This program reads a sequential stream of 8-bit ASCII characters; these characters are represented as natural numbers in the range $0 \dots 255$. We assume that *end of file* (*eof*) is $\langle RETURN \rangle$, so the program will read the input stream

$$char_1, char_2, char_3, \dots, char_k, \langle RETURN \rangle .$$

In ASCII, $\langle RETURN \rangle = 10$. The value of k is unknown until the program ends. We use the notational convenience of multiplication in this example for clarity. A macro for multiplication and its transformation into our language is given at the end of this section. This macro could be invoked with the macro statement $y := MULT(y, thousand)$ with $thousand = 1000$, and the register list changed to reflect this.

```

macro Example2(registers :  $y, x, eof, z, count$ );
 $z := x$ ;
 $y := 0$ ;
 $count := 0$ ;
while  $x \neq eof$  do
     $count := count + 1$ ;
     $y := y * 1000$ ;
     $y := y + z$ ;
     $z := x$ ;
od

```

This simple program works by concatenating the current value in the input register onto the right end of the string held in the output register by first multiplying that value by 1000 and then adding the new value. No *a priori* bound can be defined for this loop because the number of characters read (the final value of *count*) will be unknown until the program terminates. None of the current methods based on the *wp* predicate transformer is able to verify the correctness of this loop because it is unbounded. Since the loop has no *a priori* upper bound, Nelson, Hesselink, and de Bakker would have to assign the postcondition *FALSE* to this loop. We use our method to say much more than *FALSE* in a proof that it is correct.

Proof of the unbounded loop

$\{Pre : count = 0 \wedge y = 0\}$

```

{I : 0 ≤ z ≤ 255}
{Bound : none}
while x ≠ eof do
  count := count + 1;
  y := y * 1000;
  y := y + z;
  z := x;
  {R' : y = y * 1000 + xcount}
od
{R : x = eof ∧ y = Σ : 0 ≤ i ≤ count : y = y * 1000 + xi}

```

1. Show the invariant I is true before the loop executes:

- $wpw("z := x", 0 \leq z \leq 255)$
- $TRUE$

2. Show that an execution of the guarded command terminates with the invariant and the specification of the code within the loop (R') true.

- $wpw("count := count + 1; y := y * 1000; y := y + z; z := x", 0 \leq z \leq 255 \wedge y = y * 1000 + x_{count}))$
- $wpw("count := count + 1; y := y * 1000; y := y + z", wpw("z := x", 0 \leq z \leq 255 \wedge y = y * 1000 + x_{count}))$
- $wpw("count := count + 1; y := y * 1000", wpw("y := y + z", 0 \leq z \leq 255 \wedge \omega))$
- $wpw("count := count + 1", wpw("y := y * 1000", 0 \leq z \leq 255 \wedge \omega))$
- $wpw("count := count + 1", 0 \leq z \leq 255 \wedge \omega)$
- $0 \leq z \leq 255 \wedge y = y * 1000 + x_{count}$. Implied by the invariant.

3. Show that if termination occurs as we expect the postcondition is true, i.e. show that $I \wedge R' \wedge \neg(\mathcal{G}) \Rightarrow R$.

- $0 \leq z \leq 255 \wedge y = y * 1000 + x_{count} \wedge \neg(x \neq eof) \Rightarrow x = eof \wedge y = \Sigma : 0 \leq i \leq count : y * 1000 + x_i$
- $0 \leq z \leq 255 \wedge y = y * 1000 + x_{count} \wedge x = eof \Rightarrow x = eof \wedge y = \Sigma : 0 \leq i \leq count : y * 1000 + x_i$
- $x = eof \wedge y = \Sigma : 0 \leq i \leq count : y * 1000 + x_i$

4. Show $wp_i(A, R'_i), i = 1, 2, \dots$

- **Basis:** Index = 1. Then we have executed the loop one time and $count = 1$.
By step 2 we have $y = y * 1000 + x_1$
- **Hypothesis:** Assume for index $k > 1$ we have $count > 1$ and $y = y * 1000 + x_{count}$, i.e., the computation is correct to this point.
- **Show:** Show that the hypothesis holds for index $k + 1$.
 - **Case A:** $x = eof$. Then termination has occurred and by step 3 and our hypothesis we have that $x = eof \wedge y = \Sigma : 0 \leq i \leq count : y = y * 1000 + x_i$ because count was not incremented.
 - **Case B:** $x \neq eof$. Then the computation has advanced one more step and by step 2 we have that $y = \Sigma : 0 \leq i+1 \leq count+1 : y*1000+x_{i+1}$.

We now give code that could constitute a macro to perform multiplication.

macro *MULT*(registers : $y, x, z, eof, count, thousand, sum$);

sum := 0;

while *thousand* $\neq 0$ **do**

sum := *sum* + *y*;

thousand := *thousand* - 1;

```

    od;
  thousand := 1000;
  y := sum;

```

To convert this macro to a code segment in our language:

1. Number the registers: $y = x_1, x = x_2, z = x_3, eof = x_4, count = x_5, thousand = x_6, sum = x_7$

2. Remove the header with the register list:

```

  x7 := 0;
  while x6 ≠ 0 do
    x7 := x7 + x1;
    x6 := x6 - 1;
  od;
  x6 := 1000;
  x1 := x7;

```

7.2.3 Proof of Correctness of a Nonterminating Computation

The program we present here computes the z^{th} prime number in ascending order, see Figure 7.4. The program is designed to be nonterminating, so no method that depends on the *wp* predicate transformer alone can be used to prove it is correct. The program uses two macros, one that performs division without remainder returning the value 1 when a divisor for the natural number being tested is found (Figure 7.5, Section 7.3), the other that checks the range $2 \leq i \leq x - 2$ for possible divisors of the natural number x (Figure 7.6). Both of these programs use bounded loops. The main macro, *Example3* (Figure 7.4) uses an unbounded loop to calculate the set of prime

natural numbers in ascending order. We prove the bounded loop in macro *divides*, use this proof to show the correctness of macro *Prime*, and then use the correctness of macro *Prime* to prove the unbounded loop in *Example3* is correct in all of its increments.

Proof of the bounded loop of macro *divides*:

Let $IF \dots FI$ stand for the decision construct.

1. Show the invariant I is true before the loop executes.

- $wpw("t := x", x > 0)$
- $x > 0$ since $x \neq 0$
- $TRUE$

2. Show that an execution of the guarded command terminates with the invariant and the specification of the code within the loop (R') true.

- $wpw("t := t + x; IF \dots FI", x > 0 \wedge (t = y \wedge z = 1) \vee (t < y \wedge z = 0))$
- $wpw("t := t + x", wpw("IF \dots FI", x > 0 \wedge (t = y \wedge z = 1) \vee (t < y \wedge z = 0)))$
- $wpw("t := t + x", x > 0 \wedge \omega)$
- $x > 0 \wedge (t = y \wedge z = 1) \vee (t < y \wedge z = 0)$. Implied by the invariant.

3. Show that upon termination the postcondition is true, i.e., show that $I \wedge R' \wedge \neg(\mathcal{G}) \Rightarrow R$.

- $x > 0 \wedge (t = y \wedge z = 1) \vee (t < y \wedge z = 0) \wedge \neg(t < y) \Rightarrow (t = y \wedge z = 1) \vee (t > y \wedge z = 0)$
- $x > 0 \wedge (t = y \wedge z = 1) \vee (t < y \wedge z = 0) \wedge \neg(t < y) \Rightarrow (t = y \wedge z = 1) \vee (t > y \wedge z = 0)$
- $(t = y \wedge z = 1) \vee (t > y \wedge z = 0)$

4. Show $wp_i(A, R'_i)$, $i = 1, 2, \dots$

- **Basis:** Index = 1. Then we have executed the loop one time and by step 2 we have $(t = x \wedge z = 1) \vee (t < x \wedge z = 0)$.
- **Hypothesis:** Assume for index $k > 1$ we have $(t = y \wedge z = 1) \vee (t < y \wedge z = 0)$.
- **Show:** Show that the hypothesis holds for index $k + 1$.
 - **Case A:** $t \geq y$. Then termination has occurred. By step 3 we have that $x > 0 \wedge (t = y \wedge z = 1) \vee (t > y \wedge z = 0)$, since the loop did not iterate again, and t was not increased.
 - **Case B:** $t < y$. Then the computation has advanced one more step, so $t = t + x$ and by 2 we have that $(t = y \wedge z = 1) \vee (t < y \wedge z = 0)$.

Proof of the bounded loop in macro *Prime*

Let $IF \dots FI$ denote the decision construct.

1. Show the invariant is true before the loop executes.

- $wpw("i := 0", 0 \leq i \leq count)$
- $0 \leq 0 \leq count$
- $TRUE$

2. Show that an execution of the guarded command terminates with the invariant and the specification of the code within the loop (R') true.

- $wpw("z := divides(d, x); IF \dots FI; d := d + 1; i := i + 1", 0 \leq i \leq count \wedge R')$
- $wpw("z := divides(d, x); IF \dots FI; d := d + 1", wpw("i := i + 1", 0 \leq i \leq count \wedge R'))$

- $wpw("z := divides(d, x); IF \dots FI", wpw("d := d + 1", 0 \leq i \leq count \wedge \omega))$
- $wpw("z := divides(d, x)", wpw("IF \dots FI", 0 \leq i \leq count \wedge \omega))$
- $wpw("z := divides(d, x)", 0 \leq i \leq count \wedge \omega)$
- $0 \leq i \leq count \wedge 3 \leq d < x \wedge ((divides(d, x) = TRUE \wedge y = 0) \vee divides(d, x) = FALSE \wedge y = 1))$. Implied by the invariant.

3. Show that when termination occurs the postcondition is true, i.e. show that $I \wedge R' \wedge \neg(\mathcal{G}) \Rightarrow R$.

- $0 \leq i \leq count \wedge R' \wedge \neg(i < count) \Rightarrow i = count \wedge ((divides(d, x) = TRUE \wedge y = 0) \vee (divides(d, x) = FALSE \wedge y = 1))$
- $i = count \wedge ((divides(d, x) = TRUE \wedge y = 0) \vee (divides(d, x) = FALSE \wedge y = 1))$.

4. Show $wp_i(A, R'_i)$, $i = 1, 2, \dots$

- **Basis:** Index = 1. Then we have executed the loop one time and $d = 3$, so by Step 2 we have that $3 \leq 3 < x \wedge ((divides(3, x) = TRUE \wedge y = 0) \vee (divides(3, x) = FALSE \wedge y = 1))$.
- **Hypothesis:** Assume for index $k > 1$ that $3 \leq d < x \wedge ((divides(d, x) = TRUE \wedge y = 0) \vee (divides(d, x) = FALSE \wedge y = 1))$.
- **Show:** Show that the hypothesis holds for index $k + 1$.
 - **Case A:** $i = count$. Then termination has occurred and by Step 3 and our hypothesis we have that $i = count \wedge ((divides(d, x) = TRUE \wedge y = 0) \vee (divides(d, x) = FALSE \wedge y = 1))$.
 - **Case B:** $i < count$. Then the computation has advanced one more step and $d = d + 1$, so by Step 2 we have $3 \leq d \leq x \wedge ((divides(d, x) = TRUE \wedge y = 0) \vee (divides(d, x) = FALSE \wedge y = 1))$.

Proof of the unbounded loop in macro Example3

Let $IF \dots FI$ denote the decision construct.

1. Show the invariant is true before the loop executes.

- $wpw("y := 0", y \geq 0)$
- $0 \geq 0$
- $TRUE$

2. Show that an execution of the guarded command terminates with the invariant and the specification of the code within the loop (R') true.

- $wpw("y := y + 1; isprime := Prime(y); IF \dots FI", y \geq 0 \wedge p_z \leq y)$
- $wpw("y := y + 1; isprime := Prime(y)", wpw(IF \dots FI, y \geq 0 \wedge p_z \leq y))$
- $wpw("y := y + 1", wpw("isprime := Prime(y)", y \geq 0 \wedge y))$
- $wpw("y := y + 1", y \geq 0 \wedge y)$
- $y \geq 0 \wedge p_z \leq y$. Implied by the invariant.

3. Show that since termination will not occur $FALSE$ is the correct postcondition.

- $y \geq 0 \wedge p_z \leq y \wedge \neg(y > 0) \Rightarrow FALSE$
- $TRUE \wedge TRUE \wedge FALSE \Rightarrow FALSE$
- $FALSE$

4. Show $wp_i(A, R'_i)$, $i = 1, 2, \dots$

- **Basis:** Index = 1. Then we have executed the loop one time, $y = 1$, $z = 0$ or $z = 1$, so by Step 2 we have $p_z \leq y$.
- **Hypothesis:** Assume for index $k > 1$, with $y = k$ and $z \leq y$, that $p_z \leq k$.

- **Show:** Show that the hypothesis holds for index $k + 1$. $y + 1 = k + 1$ and $z + 1 \leq k + 1$ so $z \leq k + 1$. Then by Step 2 we have that $p_{z+1} \leq k + 1$.

This proof shows that it is possible to verify the correctness of a nonterminating loop that is defined on its input. With his method Dijkstra could say nothing about such a loop. With their methods de Bakker and Nelson could say only that the loop does not terminate since the postcondition is *FALSE*. With the availability of computational sequences and a recurrence between predicate transformers we can say that while the loop is nonterminating, each iteration produces the z^{th} prime number in ascending order.

7.3 Annotated Program Code for Examples

```

macro Example1(registers : val, m, n, index, count, i, stack, top, pushing, j, x);
{ $m \geq 0 \wedge n \geq 0$ }
val := 0;
if  $m = 0$  then
    val :=  $F0(n)$ ;
    { $m = 0 \wedge val = F0(n)$ }
else
    if  $n = 0$  then
        val :=  $F0(n)$ ;
        { $m > 0 \wedge n = 0 \wedge val = F0(0)$ }
    else
        val := 1; index :=  $m - 1$ ;
        count := 0;
        {Pre :  $m > 0 \wedge n > 0$ }
        {Inv :  $0 \leq count \leq n$ }
        {Bound :  $t = n - count$ }
        while  $count < n$  do
            val :=  $FM1(index, val)$ ;
            count :=  $count + 1$ ;
            {R' :  $1 \leq count \leq n \wedge val = count * FM1(m - 1, val)$ }
        od
        {R :  $count = n \wedge val = n * FM1(m - 1, 1)$ } since val was first set to 1
    fi
    {( $n = 0 \wedge val = F0(0)$ )  $\vee$  ( $n > 0 \wedge val = n * FM1(m - 1, 1)$ )} and
     $n * FM1(m - 1, 1)$  is  $FM1^n(m - 1, 1)$ 
fi
{( $m = 0 \wedge val = F0(n)$ )  $\vee$  ( $n = 0 \wedge val = F0(0)$ )  $\vee$  ( $n > 0 \wedge val = n * FM1(m - 1, 1)$ )}

```

When the index of the function (variable m) is zero and the argument of the function is greater than zero, we compute the base case (macro $F0(n)$) and the output (variable val) is equal to $f_0(n) = n + 1$. When the index of the function is greater than zero, but the argument equals zero, we compute the base case (macro $F0(n)$) again because of the function's behavior (see the note after the function definition). Here, the output val is equal to $f_m(0) = 1$. When both the function index and the function argument are greater than zero we compute $n * FM1(1) = FM1^n(1)$ since we initialize the output variable val to 1 before executing the bounded loop. This is equivalent to the functional specification in the second definition by cases. The first variable used by macro $FM1$ is the largest index in the family of functions used in the computation.

Figure 7.1. Example 1, A Total Recursive Function.

```

macro FM1(registers : val, m, n, index, count, i, stack, top, pushing, j, x);
i := 1; Stack := index || Stack; x := index;
{Pre : Stacki = m - 1 ∧ val ≥ 0}
{Inv : i ≥ 0 ∧ 0 ≤ x ≤ m - 1}
{Bound : none}
while i ≠ 0 do
  top := Stack; Stack := top || Stack;
  if top ≥ 1 then
    if val = 1 then
      val := F0(1);
    else
      if val = 0 then
        val := F0(1); pushing := val;
      else
        pushing := val; i := i + 1; val := 0;
      fi
      {pushing > 0}
      j := 0; i := i - 1; x := Stack; x := x - 1;
      {Pre : j = 0 ∧ pushing > 0}
      {Inv : 0 ≤ j ≤ pushing ∧ 0 ≤ x ≤ m - 1};
      {Bound : t = pushing - j}
      while j < pushing do
        Stack := x || Stack;
        i := i + 1;
        j := j + 1;
        {R' : 1 ≤ j ≤ pushing ∧ Stacki = x : i - j ≤ i ≤ i + j}
      od
      {R : j = pushing ∧ Stacki = x : i - pushing ≤ i ≤ i + pushing}
    fi
  else
    val := val + F0(1);
    {top = 0 ∧ val = val + F0(1)}
  fi
  {(j = pushing ∧ Stacki = x : i - pushing ≤ i ≤ i + pushing) ∨ (top = 0 ∧ val = val + F0(1))}
  i := i - 1; top := Stack;
  {R' : (top = 0 ∧ val = k * F0(1) : 1 ≤ k ≤ pushing) ∨ (top ≥ 1 ∧ Stacki = x : i - pushing ≤ i ≤ i + pushing)}
od
{R : i = 0 ∧ val = pushing * F0(1)} pushing * F0(1) = F0pushing(1). But this is the
result of functional iteration, since we only decrement the index (denoted by x), so
we compute fm-1(fm-2(⋯ f0(1))⋯)).

```

Figure 7.2. Major Macro for Example 1.

```

macro F0(registers : val, m, n, index, count, i, stack, top, pushing, j, x);
if n = 0 or n = 1 then
    val := n + 1;
else
    val := n + 2;
fi

```

Figure 7.3. Minor Macro for Example 1.

```

macro Example3(registers : z, y, x, isprime, t, count, d, i);
z := 0;
y := 0;
isprime := 0;
{Pre : z = 0}
{Inv : y ≥ 0}
{Bound : none}
while y ≥ 0 do
    y := y + 1;
    isprime := Prime(y);
    if isprime = 1 then
        p := y;
        z := z + 1;
    fi
    {R' : pz ≤ y} Where pz represents the zth prime
od
{R : FALSE}

```

Figure 7.4. Example 3, A Nonterminating Program.

```

macro divides(registers :  $z, y, x, isprime, t, count, d, i$ );
 $z := 0$ ;
if  $x \neq 0 \wedge y \neq 0$  then
   $t := x$ ;
  {Pre :  $x > 0 \wedge y > 0$ }
  {Inv :  $x > 0$ }
  {Bound :  $b = y - t$ }
  while  $t < y$  do
     $t := t + x$ ;
    if  $t = y$  then
       $z := 1$ ;
    fi
    {R' :  $(t = y \wedge z = 1) \vee (t < y \wedge z = 0)$ }
  od
fi
{S :  $(x \neq 0 \wedge y \neq 0 \wedge ((t = y \wedge z = 1) \vee (t > y \wedge z = 0)))$ }

```

Figure 7.5. Example 3, The Macro *divides*.

```

macro Prime(registers :  $z, y, x, isprime, t, count, d, i$ );
 $z := 0$ ;
if  $x \leq 1$  then
     $y := 0$ ;
else
     $count := x - 2$ ;
     $y := 1$ ;
     $d := 2$ ;
     $i := 0$ ;
    {Pre :  $count = x - 2 \wedge y = 1 \wedge d = 2$ }
    {Inv :  $0 \leq i \leq count$ }
    {Bound :  $t = count - i$ }
    while  $i < count$  do
         $z := divides(d, x)$ ;
        if  $z = 1$  then
             $y := 0$ ;
        fi
         $d := d + 1$ ;
         $i := i + 1$ ;
    {R' :  $3 \leq d < x \wedge ((divides(d, x) = TRUE \wedge y = 0) \vee (divides(d, x) = FALSE \wedge y = 1))$ }
    od
    {R :  $i = count \wedge ((divides(d, x) = TRUE \wedge y = 0 \vee (divides(d, x) = FALSE \wedge y = 1))$ }
    fi
    {S :  $((x \leq 1 \vee divides(d, x) = TRUE) \wedge y = 0) \vee (divides(d, x) = FALSE \wedge y = 1))$ }

```

Figure 7.6. Example 3, The Macro *Prime*.

CHAPTER 8

Conclusion

Current techniques using the *wp* predicate transformer are unable to deal with programs computing functions in the total recursive and recursive classes because they use the initial state/final state model that relies on termination in proofs of correctness.

We have given a language and a computing environment for it. We have given the denotational semantics and operational semantics for the language in terms that allow us to write programs to compute partial functions, and we have shown that these programs can compute only the results we expect because side effects are not possible.

We have also shown that as programs in our language are run they induce computational chains in our variable state space whose elements are both logical extensions of their predecessors (they are monotonic) and are faithful approximations to the computation's limit (they are continuous). That is, our computational chains are structures that contain valuable and reliable information about a computation in progress, whether it will eventually terminate or not.

We have defined a new predicate transformer as a conjunction of partial correctness and a recurrence relation between *wp* predicate transformers. This relation is the basis for our inductive proof method for program correctness.

Our results provide a mathematical foundation for reasoning about terminating and nonterminating processes, and a concrete method with which to reason about such processes.

8.1 Future Work

A basic property of the function specified in Example 1 of Chapter 6, Section 7.1 is given by a lemma found in Davis and Weyuker [26].

Lemma 8.1 $f_{n+1}(x + 1) = f_n(f_{n+1}(x))$.

Proof:

$$\begin{aligned} f_{n+1}(x + 1) &= f_n^{x+1}(1) \\ &= f_n(f_n^x(1)) \\ &= f_n(f_{n+1}(x)) \end{aligned}$$

With this lemma we can show that $f_n(x) = A(n, x)$:

$$f_{i+1}(x + 1) = f_i(f_{i+1}(x));$$

and

$$A(i + 1, x + 1) = A(i, A(i + 1, x)).$$

For the base cases we have

$$\begin{aligned} A(i, 0) &= 1 = f_i(0) \\ A(0, x) &= \begin{cases} x + 1 & \text{if } x = 0, 1 \\ x + 2 & \text{if } x > 1. \end{cases} = f_0(x) \end{aligned}$$

The recursively specified function $A(i, x)$ is a variant of a function introduced by W. Ackermann.

Unbounded looping and general recursion are equivalent methods for computing functions in the recursive class. Both methods work with families of functions.

In this example we have related the indices of a specific family of functions in a one-one fashion to two structurally equivalent but syntactically different functional specifications. Is the relationship shown by Davis and Weyuker an exception to or an example of a rule that can be used to relate a compositional syntax to a general recursive syntax? That is, is there a consistent system of transformations between recursively specified and compositionally specified (but structurally equivalent) functions? If such a system exists, we can use our proof technique on the version expressed compositionally, apply the transformation, and be sure that the recursive version is also correct.

A second issue deals further with recursive syntax in programming languages. The computational model we have used is deterministic and so continuous. The model Dijkstra used was nondeterministic, but because he insisted on bounded nondeterminism his model was also continuous. In both cases this makes sense because in models that use looping constructs an unbounded number of alternatives within the loop is not computationally possible. On the other hand bounded nondeterminism is a detriment when the computational model is recursive syntactically, because the requirement of continuity prohibits unlimited backtracking. In his paper Nelson [6] was careful to use a limit theorem that did not require continuity so that he could apply his model to partial functions (backtracking may not be bounded in a nonterminating computation). This question was also addressed by Hesselink [12], who showed that for syntactical recursion the Law of the Excluded Miracle holds, so continuity can be maintained in his model.

This raises two questions. First, can our *wpw* predicate transformer be conformed to a model that relaxes the continuity requirement somewhat. Because it defines a recurrence relation between predicate transformers in a repetitive situation we have

subchains of a computational chain at our disposal to investigate the computation's behavior. Is continuity of a subchain sufficient for recursive computations? If the answer is yes, then we ask whether a multiple induction scheme can be used in verification proofs involving the *wpw* predicate transformer.

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Inc., 1976.
- [2] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, pp. 453–457, August 1975.
- [3] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, pp. 576–583, October 1969.
- [4] A. R. Meyer and D. M. Ritchie, "Computational complexity and program structure," IBM research paper RC-1817, IBM, Yorktown Heights IBM Watson Research Center, 1967.
- [5] J. W. de Bakker, *Mathematical Theory of Program Correctness*. Prentice-Hall International, 1980.
- [6] G. Nelson, "A generalization of Dijkstra's calculus," *ACM Transactions on Programming Languages and Systems*, vol. 11, pp. 517–561, October 1989.
- [7] S. C. Kleene, *Introduction to Meta Mathematics*. Princeton, NJ: D. Van Nostrand, 1952.
- [8] E. C. R. Hehner, "do considered odd: A contribution to the programming calculus," *Acta Informatica*, vol. 11, pp. 287–304, 1979.
- [9] A. R. Meyer and D. M. Ritchie, "The complexity of loop programs," in *Proceedings of the ACM National Meeting*, pp. 465–469, 1967.
- [10] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.
- [11] E. C. R. Hehner, "Predicative programming part I," *Communications of the ACM*, vol. 27, pp. 134–143, February 1984.
- [12] W. H. Hesselink, "Predicate-transformer semantics of general recursion," *Acta Informatica*, vol. 26, pp. 309–332, 1989.

- [13] E. W. Dijkstra and A. J. M. van Gasteren, "A simple fixpoint argument without the restriction to continuity," *Acta Informatica*, vol. 23, pp. 1–7, 1986.
- [14] R. W. Floyd, "Assigning meanings to programs," in *Proceedings of the American Mathematical Society Symposia in Applied Mathematics*, vol. 19, pp. 19–31, 1967.
- [15] I. Greif and A. R. Meyer, "Specifying the semantics of WHILE programs: A tutorial and critique of a paper by Hoare and Lauer," *ACM Transactions on Programming Languages and Systems*, vol. 3, October 1981.
- [16] H. D. Mills, "The new math of computer programming," *Communications of the ACM*, vol. 18, pp. 43–48, January 1975.
- [17] M. H. V. Emden and R. A. Kowalski, "The semantics of predicate logic as a programming language," *Journal of the ACM*, vol. 23, pp. 733–742, October 1976.
- [18] E. Cohen, *Programming in the 1990s An Introduction to the Calculation of Programs*. Springer-Verlag, 1990.
- [19] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [20] R. Conway and D. Gries, *An Introduction to Programming*. Cambridge, MA: Winthrop Publishers, Inc., 1973.
- [21] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.
- [22] R. Sommerhalder and S. C. van Westrhenen, *The Theory of Computability: Programs, Machines, Effectiveness and Feasibility*. New York, NY: Addison-Wesley, 1988.
- [23] Z. Manna, *Mathematical Theory of Computation*. McGraw-Hill, 1974.
- [24] R. E. Davis, *Truth, Deduction, and Computation*. Computer Science Press, 1989.
- [25] A. Milli, N. Boudriga, and F. Mili, *Towards Structured Specifying: Theory, Practice, Applications*. New York, NY: Ellis Horwood Limited, 1989.
- [26] M. D. Davis and E. J. Weyuker, *Computability, Complexity, and Languages*. New York, NY: Academic Press, Inc., 1983.

MICHIGAN STATE UNIV. LIBRARIES



31293014172278