CWI Monographs 6

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

Program Correctness over Abstract Data Types, with Error-State Semantics

J.V. Tucker
J.I. Zucker

North-Holland

Program Correctness over Abstract Data Types, with Error-State Semantics

CWI Monographs

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The CWI is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.)

CWI Monograph 6

Program Correctness over Abstract Data Types, with Error-State Semantics

J. V. Tucker
J. I. Zucker

1988

North-Holland
Amsterdam · New York · Oxford · Tokyo

ISBN: 0 444 70340 3

Publishers:
Elsevier Science Publishers B.V.
P.O. Box 1991
1000 BZ Amsterdam
The Netherlands

Sole distributors for the U.S.A. and Canada:
Elsevier Science Publishing Company, Inc.
52 Vanderbilt Avenue
New York, N.Y. 10017
U.S.A.

Cover: Tobias Baanders

Printed in the Netherlands

To our parents

Preface

This book is a research monograph in the theory of program specification and verification. More specifically, it is about Hoare-style proof systems for proving the correctness of programs implemented over abstract data types. In addition, the proof systems are designed to operate on programs with the semantical feature that using an uninitialized variable leads to an error message.

Our objective is to analyse mathematically the proof systems in the manner of the monograph *Mathematical theory of program correctness* (1980) by J.W. de Bakker. We give a careful treatment of the many semantical and logical issues involved to support rigorous proofs of the soundness and completeness of the proof systems. Interestingly, much of the semantical machinery used by De Bakker adapts well to the more general setting, but several innovations are required in the logical foundations of the proof systems. A full discussion of the many themes to be found in the book is made in the Introduction (see, for example, the précis in Section 0.2). We hope other researchers in the field will find several useful technical ideas which can be employed in their own work.

The principal ideas for this monograph came in 1979 while the authors were at the CWI (Centre for Mathematics and Computer Science, formerly the Mathematical Centre), Amsterdam. Our cooperation was organized so that the first author (JVT) was mainly responsible for the Introduction and Chapter 4, and the second author (JIZ) for Chapters 1, 2 and 3. In overcoming the considerable difficulties involved in bringing to completion such a technically involved project we received assistance from several sources, and our special thanks are due to

— Jaco de Bakker, Head of the Department of Software Technology of the CWI and editor of the CWI Monograph Series, for encouraging the project from its inception;

— the CWI, for generously providing funds and hospitality to both authors, after they had left its employ, enabling them to meet together on a number of occasions for consultation on the project;

— Bar Ilan University and the Weizmann Institute, Israel, for generously providing funds and hospitality to the first author during the summer of 1981, while the second author was working at Bar Ilan University;

— Leeds University for providing hospitality to the second author during the summer of 1986;

— and the National Science Foundation, whose grants to the second author (MCS-8305426 and DCR-8504296) provided funds for equipment for text preparation, and travel for both authors.

Many people read the text while it was in preparation, and provided useful suggestions, in particular Jonathan Stavi (Bar Ilan University), Jan Terlouw (University of Utrecht), Jan Bergstra (Universities of Amsterdam and Utrecht), Clive Jervis and Benjamin Thompson (Leeds University), and Pierre America (Philips Research Laboratories, Eindhoven).

The book was prepared in camera-ready form at the University of Buffalo, using the Ditroff text processor on the UNIX operating system, and a QMS Lasergrafix 1200 laser printer. The text was typed into the system by the second author, except for Chapter 4, which was typed in by Lynda Spahr. Invaluable hardware and software support were provided by Robert Coggeshall, John LoVerso and Don Gworek. The improvement of fonts, and design of special characters, were performed admirably by George Sicherman, Scott Mesches and John Arrasjid.

John Tucker
Jeffery Zucker

July 1987

Table of Contents

Chapter 0

Introduction

Have nothing in your houses that you do not know to be useful, or believe to be beautiful.

William Morris
Hopes and Fears for Art

Hostinato rigore.

Leonardo da Vinci
Windsor Notebooks

0.1 BACKGROUND

The mathematical theory of program correctness is founded upon two ideas of independent and contemporary origins:

FLOYD-HOARE THESIS: *Formally specified programs can be formally verified in logical systems designed around the syntactic structure of the programming language; indeed such logical systems may be used to specify the semantics of the language* (R.W. Floyd and C.A.R. Hoare).

SCOTT-STRACHEY THESIS: *Programs can be considered as mathematical objects having a mathematically exact syntax and semantics* (D.S. Scott and C. Strachey).

The two ideas have been enormously influential in the field of programming methodology and programming language design. The work of Floyd and Hoare initiated a vigorous development of program proving techniques with an emphasis on how syntactic logical tools can determine program meanings: the language definition method of *axiomatic semantics* (see Floyd [1967], Hoare [1969], and Hoare and Wirth [1973]). The work of Scott and Strachey provided considerable conceptual understanding of programming languages by means of the mathematical tools of *denotational*

semantics for the definition of programming languages (see Scott [1970] and Scott and Strachey [1970]).

The conjunction of the two ideas came with the realization that the reliability of a proof system for correctness in a programming language depends upon a rigorous and independently defined semantics for the language, to complement the axiomatic semantics. This reliability is expressed by a *soundness theorem* wherein the logical system is shown to generate valid statements about the programming language, equipped with the independent semantics. A soundness theorem demonstrates that the proof system is indeed consistent. In addition, the independent semantics provides an opportunity to try to prove a *completeness theorem* wherein the logical system is shown to generate *all* the valid statements of the requisite kind about the programming language.

The first mathematical analysis of a logical system for program verification occurred in Cook [1976], later published as Cook [1978]. Here the mathematical structure of logics for program correctness was carefully explored, and the soundness and completeness of such systems seriously discussed and settled (almost: see De Bruin [1984] and the corrigendum to Cook [1978], and Bergstra and Tucker [1982a, 1982b]). In addition, a number of technical concepts related to completeness were presented.

In many respects, the present-day mathematical theory of program correctness is largely the result of developing the situation described in Cook [1978] in three directions:

(1) the construction of proof systems for more elaborate programming language features;
(2) the use of proof systems as an instrument to experiment with the semantics of programming language constructs; and
(3) the study of the metamathematics of proof systems for program verification.

Certainly, the pace of progress in recent years has quickened, and enough has been achieved to establish the theory as an essential tool for programming language design.

A concise introduction and survey of the theory of correctness for deterministic sequential programs is Apt [1981]; and for information on nondeterministic and concurrent constructs see Apt [1984]. For a detailed and rigorous treatment of the basic programming language constructs there is, however, no substitute for De Bakker [1980]. Here the true mathematical complexity of the soundness and completeness proofs are confronted; in addition the book contains a detailed description of the evolution of the subject.

0.2 AIMS

In this monograph we present a new mathematical treatment of the soundness and completeness of proof systems for the partial correctness, and total correctness, of programs made with some basic constructs, namely: *iteration and recursion in the presence of arithmetic, booleans and arrays.* Our starting point is the account of iteration, recursion and arrays given in De Bakker [1980] (Chapters 1—5 and the Appendix) which we extend in both of the directions numbered (1) and (2) in the previous section; as we shall now explain.

0.2.1 Programming language constructs

We consider computations not just by programs allowing a single kind of variable — specifically the integers in De Bakker [1980] — but computations by programs involving a finite number of syntactically distinct kinds of variables of arbitrary type, only one of which may be numerical. In particular, two basic features of contemporary programming languages, not covered in De Bakker [1980], simultaneously receive attention here:

(i) *Abstract data types.* The precise nature of the data types appearing in programs is not fixed in advance by the programming language definition; the names of the types and their primitive operators are fixed for each program, however. We do not allow programs to create data types and we do not consider the mechanisms by which these types are defined. Rather our work treats the correctness theory of programs at any given level of data abstraction. We assume the data types for each program are given semantically as a *class* $\mathbb{K}$ *of many-sorted algebras on which the program is interpreted.*

The advantage is that it is immaterial to our theory whether a data type class $\mathbb{K}$ represents a family of implementations of an axiomatic specification (as one finds in Hoare [1969] or in the algebraic theory of Liskov and Zilles [1975], ADJ [1977], Guttag and Horning [1979]) or represents the semantics of a data type module (as one finds in programming languages such as ALPHARD, CLU, MODULA-2 or ADA (Wulf, London and Shaw [1976], Liskov *et al.* [1981], Wirth [1983], Ada [1983] and Ichbiah [1983]). In allowing for data abstraction we are allowing for the effects of a new programming language feature on program correctness without commitment on the form of the feature. A detailed discussion of our treatment of abstract data types follows in Section 0.3; we presume the reader is acquainted with the subject, and, in addition to the papers and books already mentioned, recommend Wulf [1980].

(ii) *Many-typed programs.* The programs considered are multityped in the simple sense that they may contain more than one type of variable. This modest extension of the language features previously considered in the theory of program correctness is noteworthy because it is not at all a trivial matter as far as the completeness of proof systems is concerned. As will be explained in Section 0.5, the standard treatment of completeness for iteration in a single type language breaks down when generalized to two or more types.

0.2.2 Semantics of language constructs

In our setting of abstract data types we redesign the semantics of our basic constructs in order to model *errors* which may arise in a computation from uninitialized variables and which cause the computation to halt in an error state. This error semantics is very well known in practice, and has been considered in connection with automatic program verification (German [1978]). However, the semantics receives its first mathematical analysis here. (Incidentally, the error semantics supports an exception mechanism for the languages, but the implications of this observation are not fully pursued.)

The states of a computation are only *partial functions* from variables to values, and at any stage in a computation at most finitely many variables have had values assigned to them. All other variables have a unique *unspecified value* ω associated with them, and *calling* such an uninitialized variable in the course of a computation will cause the computation to halt in an *error state* ε. Any expression containing a variable of unspecified value may itself, on evaluation, gain the unspecified value (see Section 1.2). In particular, considering expressions of boolean type, we are led to a three-valued logic for tests in control structures. The details of this extension to error semantics, its effect on specified programs, and its connection with exception handling, are presented in Section 0.4.

0.2.3 Metamathematics

The two extensions of language features described in 0.2.1 require us to extend the ideas used to frame and prove the completeness theorems for the new proof systems; thus, this monograph also extends De Bakker [1980] in the metamathematical direction numbered (3) in Section 0.1.

Briefly stated, the extensions to abstract types and many-typed programs force us to abandon the usual *first-order assertion language* and its associated concept of the *completeness of a proof system in the sense of Cook*. The assertion language is said to be *expressive* for a programming language

on an interpretation if all weakest preconditions (or, equivalently, strongest postconditions) of the programs relative to the interpretation are definable in the assertion language. Completeness in the sense of Cook is the property that whenever the assertion language is expressive on an interpretation then the valid specified programs are all provable.

In the single-type situation, the first-order assertion language is expressive for many programming features interpreted on the integers (see Zucker [1980]). But it is not expressive on many other structures even for iteration (see Wand [1978], Bergstra and Tucker [1982a], Clark, German and Halpern [1983]). Furthermore, *the many-sorted first-order assertion language is not expressive for iteration on most many-sorted structures* (see Bergstra and Tucker [1984]). The result of this fact is that *the generalization of Cook's work to many-typed programs fails to find applications if a first-order assertion language is used.*

In this monograph we use a stronger assertion language, essentially a weak second-order language, which enables us to prove expressiveness for our programming constructs relative to any given class of interpretations in a natural uniform sense. Furthermore, the new assertion language enables us to embrace total correctness, which is, in a certain sense, beyond the scope of a first-order assertion language (see Apt [1981]).

0.2.4 Computability in an abstract setting

The construction of the formulae expressing computation by the programs requires substantial theoretical work involving a theory of families of functions that are computable by programs on an abstract data type. This theory is based on notions of inductive definability that generalize, to a data type class, inductive definitions of the partial recursive functions on the natural numbers. It is shown that two classes of inductively definable functions coincide with the two classes of functions computable by **'while'** programs with arithmetic, and with and without arrays.

This new characterization is of interest in its own right. For example, it is relevant to studies of the power of programming features and to studies of the semantics of function procedures in programming languages. It also supports a certain Generalized Church-Turing Thesis concerning effectively calculable functions in an abstract setting.

In the next three sections we will discuss these ideas in greater depth and, after this Introduction, we will develop the subject in the systematic and austere fashion that becomes a mathematical theory. We conclude this section on objectives with some statements about rigour.

0.2.5 Rigour

It is an aim that this monograph provide a self-contained, compact and rigorous account of its constructs, with all but the truly routine and tedious details excluded. We hope that experts will have no difficulty in extracting the many new devices it contains, and that less experienced readers will find all points adequately explained.

The value of formal studies of programming language constructs to the computer science community at large depends on their correctness. It is essential that proper standards of mathematical rigour be established wherever possible: standards that are *adhered to* by the theorist and *appreciated* by the experimentalist. The emergence of the theory of program correctness, sketched in Section 0.1, was plagued by unsound proof rules and improper semantic definitions: formal errors that signal fundamental problems of rigour. We believe that it is one of the significant achievements of De Bakker [1980] that many of the difficulties in making semantics and proving soundness and completeness are overcome. By using De Bakker [1980] as a blueprint, and documenting our extensions and deviations, we hope to confirm in our turn that rigour — *obstinate rigour*, in Leonardo's motto — is robust and indispensable.

0.3 DATA TYPE SEMANTICS

We suppose that each data type ***dt*** appears in a program as a many-sorted signature Σ whose sorts name types in variable declarations and whose function symbols name the basic operations on data allowed in assignments and tests. Thus, syntactically, ***dt*** is represented by Σ. In a programming language with data type modules, a signature Σ is essentially a module heading. Let ***Statemt***(Σ) be a set of programs invoking ***dt***.

Semantically, the data type ***dt*** is represented by a class $\mathbb{K}$ of algebraic structures of signature Σ. Each structure $\mathbb{K}$ represents a collection of admissible implementations of ***dt***. Thus, for a program $S \in$ ***Statemt***(Σ) invoking ***dt*** we will define its semantics $\mathcal{M}(S)$ as a *family* of transformations of computation states

$$\mathcal{M}_A(S)\colon \mathrm{STATE}(A) \to \mathrm{STATE}(A)$$

indexed by $A \in \mathbb{K}$. Our attention will always be concentrated on computational properties of the program S that are uniform over all implementations of ***dt*** in $\mathbb{K}$. In particular, the correctness of specified programs over ***dt*** will be defined by the correct specification of their behaviour on all implementations in $\mathbb{K}$. Let us illustrate the use of the class $\mathbb{K}$, first in

connection with the traditional specification of data types in program language definitions, and secondly, and more importantly, with the programming features of representation-independent and user-defined data types in contemporary programming languages.

0.3.1 Standard types

The types *integer*, *boolean*, *real* and *character* appear in a host of languages descended from Fortran and Algol 60. For each language there is a set of familiar operations described in the syntactic specification of the language. What forms do the semantics of these types have?

First, there is a class $\mathbb{K}$ containing a single four-sorted structure A whose domains implement the four basic types. The structure A may be a mathematical semantics involving the infinite ring of integers and the infinite field of real numbers; or A may be one of several finite machine approximations to a mathematical semantics. Such a $\mathbb{K}$ may be of general interest if A is a mathematical semantics designed to idealize the types for a wide audience (as in De Bakker [1980]); or $\mathbb{K}$ may be tailored to the particular users of a particular machine implementation A of the types.

However, it is unlikely that a programming language would have its ground types so constrained: for instance, the numerical size of MAXINT does not belong to a language definition. Thus, a more practical characterization requires a class $\mathbb{K}$ of distinct implementations satisfying certain properties. A formal and axiomatic approach to the definition of the standard types was first taken in Van Wijngaarden [1966], and re-examined in the context of program verification in Hoare [1969].

0.3.2 Data Abstraction

An idea central to the theory of data types is the following:

DATA ABSTRACTION PRINCIPLE. *Let A and B be Σ-structures representing two implementations of data type* ***dt****. Then A and B are considered to represent identical implementations of* ***dt*** *if, and only if, A and B are algebraically isomorphic.*

The principle defines what is abstract about an abstract data type: the method used to represent data in an implementation is suppressed by considering the implementation to be uniquely determined up to isomorphism only. The abstract semantics $\mathbb{K}$ of a data type ***dt*** based on an implementation A is the *isomorphism type* $\mathrm{ISO}(A)$ of A, i.e. the class of all Σ-structures isomorphic to A: see ADJ [1977] and Wulf [1980].

Now any program semantics that is designed to operate with abstract data types must satisfy the following condition:

PROGRAM SEMANTICS ABSTRACTION PRINCIPLE. *Let A and B be Σ-structures representing two implementations of data type* ***dt***. *Suppose that A and B are isomorphic by $\varphi: A \to B$. Then φ determines a state space isomorphism $\hat{\varphi}$: STATE$(A) \to$ STATE(B) such that for every $S \in$* ***Statemt***(Σ) *the following diagram commutes:*

$$\begin{array}{ccc} \text{STATE}(A) & \xrightarrow{\mathcal{M}_A(S)} & \text{STATE}(A) \\ \Big\downarrow \hat{\varphi} & & \Big\downarrow \hat{\varphi} \\ \text{STATE}(B) & \xrightarrow{\mathcal{M}_B(S)} & \text{STATE}(B) \end{array}$$

Thus for each $\sigma \in$ STATE(A)

$$\hat{\varphi}(\mathcal{M}_A(S)(\sigma)) = \mathcal{M}_B(S)(\hat{\varphi}(\sigma)).$$

In consequence, *the semantics of a program S on a data type* ***dt*** *is uniquely determined by the isomorphism type representing the abstract data type semantics of* ***dt***.

The Program Semantics Abstraction Principle is an essential companion of the Data Abstraction Principle; unfortunately, it is invariably neglected in the literature. (For some of the implications of this principle see Tucker [1980].)

0.3.3 General types

The facility of modules for user-defined types in programming languages, and the formal methods for their specification, introduce a great variety of classes that arise as data type semantics.

The use of axioms to define the built-in types of a language, mentioned earlier in 0.3.1, readily applies to data types in general. In its most general form, axiomatic specification amounts to defining the semantics of a data type as some class $\mathbb{K}$ of *acceptable implementations* whose operations are named by Σ and satisfy properties recorded as a set T of axioms. The pair (Σ,T) is called a *data type specification* and the axioms of T are written in some formal language such as a first-order logical language. Thus $\mathbb{K}$ is a subclass of MOD(Σ,T), the class of all Σ-models satisfying T.

What more can be said about the qualification "acceptable implementations"?

As we have explained in 0.3.2, the theory of data types employs the Data Abstraction Principle, and so the classes of interest in that theory are closed under isomorphism:

$$A \in \mathbb{K} \quad \text{implies} \quad \mathrm{ISO}(A) \subseteq \mathbb{K}.$$

Next, in the theory of data types, a data type implementation is represented by a special kind of structure A satisfying the property that A is generated by elements named in its signature: such structures are called *minimal algebras* in ADJ [1977]. This minimality condition ensures that the entire type can be accessed from basic data by applying the basic operations. In addition, the condition ensures that each element of A can be named by a syntactic expression over Σ.

Let $\mathrm{MIN}(\Sigma,T)$ be the class of all minimal models of T; then the class $\mathbb{K}$ is expected to be a subclass

$$\mathbb{K} \subseteq \mathrm{MIN}(\Sigma,T) \subset \mathrm{MOD}(\Sigma,T).$$

There are further theoretical constraints that can be imposed, the most obvious of which is that a data type be implementable, in a sense consistent with the Church-Turing Thesis. This means that each $A \in \mathbb{K}$ ought to be an *effectively computable structure:* in an obvious notation

$$\mathbb{K} \subseteq \mathrm{MIN}(\Sigma,T) \cap \mathrm{COMP}(\Sigma,T) \subset \mathrm{MOD}(\Sigma,T).$$

Actually there are three kinds of effectively calculable structure of importance in the theory of data types: *computable, semicomputable* and *cosemicomputable structures,* corresponding to the concepts of recursive, r.e. and co-r.e. in computability theory. For their rigorous definition and principal properties, see Bergstra and Tucker [1982d, 1983b] and Meseguer and Goguen [1985].

Returning to the subject of formal specification, in the case of user-defined types it is common to identify the semantics of a data type as the isomorphism type of a specific structure, and for this task the *algebraic specification methods* are well suited.

Suppose that the axioms in T are algebraic: equations or conditional equations to be specific. Then it is possible to define the semantics of the specification (Σ,T) as the isomorphism type of the initial algebra $I(\Sigma,T)$ in $\mathrm{MOD}(\Sigma,T)$; the structure $I(\Sigma,T)$ is a semicomputable minimal algebra if T is an r.e. axiomatization. Furthermore, it may also be possible to choose the final algebra $F(\Sigma,T)$ in $\mathrm{MOD}(\Sigma,T)$ for the same purpose; if T is r.e. and $F(\Sigma,T)$ exists then $F(\Sigma,T)$ is a cosemicomputable algebra; see Bergstra and Tucker [1982d, 1983b] for further details.

Notice that many of these classes are not axiomatizable in first-order logic.

0.3.4 Arithmetic

The special properties of the classes that arise as data type semantics are *not* required in the mathematical theory of program correctness that follows. The classes of interest are not quite free of hypotheses, however.

In addition to user-defined types, our programs will involve arithmetic and booleans. The presence of these special types entails that our programs are interpreted on structures with distinguished domains for the natural numbers $\mathbb{N}=\{0,1,\dots\}$ and boolean truth values $\mathbb{B}=\{\mathrm{t},\mathrm{f}\}$. For want of a better name we call such structures *standard structures*. Thus *the mathematical theory will proceed from the single assumption on a class* $\mathbb{K}$, *that it contain only standard structures.*

If $\mathbb{K}$ is any class of structures it may be "standardized" as follows.

Let A and B be any structures with disjoint signatures Σ_A and Σ_B. We can define the *join* $[A,B]$ of A and B to be the structure with signature $\Sigma_A \cup \Sigma_B$ whose domains and operations are those of A and B. Let $\mathbb{N}$ denote the standard model of arithmetic and $\mathbb{B}$ the standard two-element boolean algebra. Then an arbitrary structure A can be standardized by making $[A,\mathbb{N},\mathbb{B}]$, and for any class $\mathbb{K}$ we can set

$$S(\mathbb{K}) = \{[A,\mathbb{N},\mathbb{B}] \mid A \in \mathbb{K}\}.$$

0.4 ERROR STATE SEMANTICS AND CORRECTNESS

0.4.1 Program semantics

A *proper state* of a computation on a structure A is a function σ that assigns to each variable of the programming language either an element of A or a distinguished *unspecified value* u, where, moreover, σ takes the unspecified value u for all but finitely many variables. There is also a special *error state* ε. Let PR.STATE(A) be the set of all proper states over A, and let STATE(A) be the set of all states over A.

The meaning of a program S on a structure A is a partial function

$$\mathcal{M}_A(S)\colon \text{PR.STATE}(A) \to \text{STATE}(A)$$

called a *state transformation.* When this state transformation is applied to a proper state σ there are three possible outcomes:

(1) The execution of S with initial state σ *terminates in a proper state* σ'; in symbols $\mathcal{M}_A(S)(\sigma)\downarrow\sigma'$ or $\mathcal{M}_A(S)(\sigma)=\sigma'$. Here we say that the computation *converges normally* or simply *converges.*

(2) The execution of S with initial state σ *terminates in the error state* ε; in symbols $\mathcal{M}_A(S)(\sigma)\downarrow\varepsilon$ or $\mathcal{M}_A(S)(\sigma)=\varepsilon$. Here we say that the computation *converges exceptionally* or simply *aborts.*

(3) The execution of S with initial state σ *does not terminate:* $\mathcal{M}_A(S)(\sigma)$ is not defined; in symbols $\mathcal{M}_A(S)(\sigma)\uparrow$. Here we say that the computation *diverges.*

A computation aborts only if the program required the evaluation of an expression containing a variable in a state where that variable had the unspecified value $\mathbb{u}$.

The meaning of a program S on a class $\mathbb{K}$ is a family of partial functions

$$\mathcal{M}_{\mathbb{K}}(S)=\{\mathcal{M}_A(S)\mid A\in\mathbb{K}\}.$$

0.4.2 Specified programs

We will study Hoare-like proof systems for programs operating with this semantics. As usual, a *specified* or *asserted program*, or *correctness formula*, has the syntactic form

$$\{p\}S\{q\}$$

where p and q are formalized statements or assertions about the values of the variables in S; p is called the *input condition* or *precondition* and q is called the *output condition* or *postcondition.* However the new program semantics materially alters the meaning of the correctness formulae.

Consider first *partial correctness* semantics. The formula $\{p\}S\{q\}$ is true in a proper state σ over a structure A if, and only if, the following condition is met:

> *if* p *holds at* σ *and the execution of* S *starting at* σ *terminates in some state* σ', *then* σ' *is not the error state* ε *and* q *holds at* σ'.

Notice that a sharp distinction is made between computations that diverge and those that abort.

Consider now *total correctness* semantics. According to this, the formula $\{p\}S\{q\}$ is true in σ over A if, and only if, the following holds:

> *if* p *holds at* σ, *then the execution of* S *starting at* σ *terminates in some state* σ' *that is not the error state and* q *holds at* σ'.

The formula $\{p\}S\{q\}$ is (in both cases) said to be true on a structure A if, and only if, it is true of every proper state over A. Finally, it is true in a class $\mathbb{K}$ if, and only if, it is true on every structure $A \in \mathbb{K}$.

We are interested in the specification of program behaviour for such a class $\mathbb{K}$ of data type implementations, and its formal verification.

0.4.3 Example

Informally, let us call a program S *semantically closed* (for want of a better term) if every variable in S, called in any computation of S, is first initialized by S. For example, the following program is semantically closed:

$$
\begin{aligned}
S \equiv\ & x := 0; \\
& \textbf{if true then } y := x \\
& \qquad \textbf{else } x := y \\
& \textbf{fi}
\end{aligned}
$$

But S is not what one would call syntactically closed (meaning: all variables appearing in the program are initialized).

With the partial correctness semantics of the correctness formulae, we are able to give a formal definition of the idea. Let S be a program with variables $\{v_i \mid i \in I\}$. Then S is *semantically closed* for data type class $\mathbb{K}$ if the correctness formula

$$\{\mathbin{\mathbb{A}}_{i \in I}\ v_i = \textbf{unspec}\}S\{\textbf{true}\}$$

is valid for the class $\mathbb{K}$; here **unspec** is a name for the unique unspecified value $\mathbb{u}$.

Interestingly, notice that the formula

$$\{\textbf{true}\}S\{\textbf{true}\}$$

is valid for precisely the semantically closed programs: with the conventional, error-free partial correctness semantics in De Bakker [1980] it is valid for all programs, of course.

0.4.4 Errors and exception handling

The calling of an uninitialized variable is a runtime property of a program that may warrant an exception handling mechanism. The semantics used here is relevant to a formal study of exception handling, possibly starting from the methodological studies of Goodenough [1977] and Liskov and Snyder [1979] (see also Cristian [1983] and Luckham and Polak [1980]).

To see this, recall that the programs we study belong to a modularized programming language, and suppose that S is a procedural module used by another program module C, the *caller* of S. The program C *invokes* S on a state σ, and in the course of the execution or *activation* of S on σ the error state ε may be encountered and interpreted as a signal to C. The caller C now *raises the exception* that an uninitialized variable is used, and a program H, the *exception handler*, is called to deal with the exception.

The study of the error or exception semantics for the activations of S can be separated from the study of the models for the exception mechanisms, which address the relationship between C, S and H. For example, with reference to Goodenough [1977] and Liskov and Snyder [1979], their resumption and termination models are both consistent with our error semantics. We note (in connection with Liskov and Snyder [1979]) that a resumption model would seem appropriate here, for on discovering an uninitialized variable the user may be invited by the handler to initialize it and continue.

0.4.5 Assertion languages

The precondition p and postcondition q in the specified program $\{p\}S\{q\}$ will be taken from a formally defined *assertion language.* The assertion languages of interest will be discussed shortly: here we just wish to point out one aspect of them. For each type in the programming language, the assertion language will contain *two* kinds of variable:

(i) program variables or identifiers belonging to the programming language; and
(ii) special assertion language variables.

Now quantification in the assertion language is allowed only over assertion variables. This means that, typically, assertions have to be evaluated at a pair (ρ,σ) where

(a) σ is a proper state over the interpretion A; and
(b) ρ is a valuation of the assertion variables, i.e. a function from assertion variables to values from $A \cup \{\mathbb{u}\}$.

Although boolean tests in the control structures of the programming language have three-valued semantics, and are allowed as subexpressions within assertion formulae, the assertion language will have a conventional two-valued semantics.

0.5 COMPLETENESS

We will present proof systems for the partial correctness and total correctness of programs, operating on an arbitrary data type class $\mathbb{K}$, and prove the soundness and completeness of these systems for the error-state semantics. These soundness and completeness theorems are proved for programming languages of increasing complexity:

(*i*) straight-line programs with arithmetic and arrays;
(*ii*) 'while' programs with arithmetic and arrays; and
(*iii*) programs with recursion, arithmetic and arrays.

Each language will be the subject of a chapter.

In proving a completeness theorem, a problem of central importance is to prove that weakest preconditions and strongest postconditions can be expressed in the assertion language, uniformly for $\mathbb{K}$, a property commonly termed *expressiveness* or *expressibility* of the assertion language. For each of the programming languages (*i*)—(*iii*) this is done in a different way.

0.5.1 Straight-line programs

This is the subject of Chapter 1, with the Soundness Theorem in Section 1.5 and the Completeness Theorem in Section 1.7

For straight-line programs with arithmetic and arrays, the weakest precondition and strongest postcondition of a program S with respect to an assertion p can be expressed by assertions $\mathsf{wp}[S,p]$ and $\mathsf{sp}[p,S]$ defined by straightforward induction on the complexity of S. It is possible to use as assertion language the first-order language $\boldsymbol{Lang}_1(\Sigma^u)$ over the signature Σ^u, being the signature Σ of the types of $\mathbb{K}$ augmented by a name for the unspecified value u.

0.5.2 'while' programs

This is the subject of Chapter 2, with the Soundness Theorem for partial correctness in Section 2.5 and the Completeness Theorem in Section 2.7; total correctness is considered in an appendix.

To define the weakest preconditions and strongest postconditions, we must first formalize the notion of a *computation sequence* for a program S, being a sequence of states starting from some initial state and, if finite, ending at the final state of a computation by S. For this, $\boldsymbol{Lang}_1(\Sigma^u)$ no longer suffices, and we consider a new language $\boldsymbol{Lang}_1(\Sigma^*)$ which allows variables for finite sequences of data. More precisely, for each $A \in \mathbb{K}$ and each sort i we adjoin a domain A_i^* consisting of all functions

$$\xi\colon \mathbb{N} \to A_i \cup \{\mathbb{u}\}$$

such that the set

$$\{n \in \mathbb{N} \mid \xi(n) \neq \mathbb{u}\}$$

is finite; and we adjoin the necessary *application operation*

$$\mathsf{Ap}_i(\xi, n) = \xi(n).$$

The resulting structure is called A^*, with signature Σ^*, and the class of all such A^* for $A \in \mathbb{K}$ is denoted $\mathbb{K}^*$. $\boldsymbol{Lang}_1(\Sigma^*)$ is the first-order language over Σ^*.

With this assertion language, the notion of (a code for) a computation sequence for a program S can be defined by induction on the complexity of S; and from this the weakest preconditions and strongest postconditions can easily be defined.

0.5.3 The failure of first-order many-sorted assertion languages

The replacement of $\boldsymbol{Lang}_1(\Sigma^{\mathbb{u}})$ by $\boldsymbol{Lang}_1(\Sigma^*)$, or some such stronger language, is essential. For even in the case of a one-sorted abstract type A and 'while' programs with arithmetic, the first-order language is not expressive; indeed in Bergstra and Tucker [1984] an example is given where A is the standard model of arithmetic! More precisely, the two-sorted first-order assertion language is not expressive for while programs on the join $[\mathbb{N},\mathbb{N}]$ (recall the join from 0.3.4). In addition, two-sorted Hoare's logic for while programs on $[\mathbb{N},\mathbb{N}]$ is incomplete.

0.5.4 Recursive programs

This is the subject of Chapter 3, with the Soundness Theorem for partial correctness in Section 3.4 and the Completeness Theorem in Section 3.7; total correctness is considered in an appendix.

For recursive programs, we again used $\boldsymbol{Lang}_1(\Sigma^*)$ to encode the notion of a *computation sequence* for a program $R = \langle D \mid S \rangle$ (where D is a sequence of procedure declarations and S is a statement). However this notion cannot be defined simply by induction on the complexity of S, and a new, more sophisticated, approach is needed. Roughly speaking, we proceed as follows:

(*a*) For a given R we show that the function which assigns a computation sequence $\tau = (\sigma_0, \sigma_1, \ldots)$ to an initial state $\sigma = \sigma_0$ can be faithfully represented by a function ψ_R on A^* which is a *function inductively definable uniformly over* $\mathbb{K}^*$.

(*b*) From results belonging to the later Chapter 4 (see below) it follows that that ψ_R is a *function computable by a* 'while' *program uniformly over* $\mathbb{K}^*$.

(*c*) From the results proved in Chapter 2, mentioned above, it can be deduced that the graph of ψ_R is *definable in* $\boldsymbol{Lang}_1(\Sigma^*)$ uniformly over $\mathbb{K}^*$.

(*d*) It is now a simple matter to obtain formulae in $\boldsymbol{Lang}_1(\Sigma^*)$ for the weakest precondition and strongest postcondition from the formulae in (*c*).

Let us comment further on the steps (*a*) and (*b*).

0.5.5 Computability and inductive definability

The functions inductively definable over a class $\mathbb{K}$ are those functions on the structures of $\mathbb{K}$ which are uniformly definable by certain *induction schemes*, including *simultaneous primitive recursion* on $\mathbb{N}$ and the *least number* or *μ-operator* on $\mathbb{N}$. Their definition arises naturally from the features of the functions used to make the operational semantics for recursive programs, but it is of interest in its own right, as a mathematical definition of a computation on $\mathbb{K}$. The induction schemes generalize, to a class $\mathbb{K}$, the inductive definition of the partial recursive functions on $\mathbb{N}$ due to S.C. Kleene [1952]. For this reason we devote Chapter 4 to the subject of inductive definability (and postpone from Chapter 3 the proof of the results required in step (*b*) above).

From the point of view of the program correctness theory, the main outcome of Chapter 4 is that the class of functions inductively definable over $\mathbb{K}$ is precisely that computed by 'while' programs with arithmetic, *but without arrays*. Interestingly, the technical fact required in Chapter 3, that inductively definable functions are computable by 'while' programs with arithmetic, is proved quite smoothly. The converse however, included for completeness of our presentation, is more cumbersome to prove because the semantics of our programs includes features (unspecified arguments, the error state) which are out of place in a (pure) mathematical theory of computation. We trust that this work will be semantically interesting because it concerns function-theoretic aspects of programs.

Chapter 4 also includes the answer to the following question: What is the inductive definability counterpart to 'while' programs with arithmetic *and with arrays*? We prove the attractive result that on replacing primitive recursion with *course-of-values recursion* in the original definition of inductive definability over $\mathbb{K}$, the resulting class of functions definable by *course-of-values induction* over $\mathbb{K}$ is precisely that computed by 'while'

programs with arithmetic and arrays.

We conclude Chapter 4 with information on these classes of functions and consider them in the context of research into computability on abstract structures. We will survey a number of disparate approaches to the definition of a computable function on an algebraic structure taken from mathematics and computer science, and discuss their equivalence and the implications for a Generalized Church-Turing Thesis. The formulation of a Generalized Church-Turing Thesis and its role in the theory of programming languages will be discussed in detail.

0.6 REMARKS FOR THE READER

0.6.1 Recommended prerequisites

Although the present work is mathematically self-contained, it is best viewed as a development of certain investigations in De Bakker [1980], and so some acquaintance with parts of that book (Chapters 1—5 and the Appendix) will be useful to the reader. In fact the notation here has purposely been kept as similar as is practicable to that of De Bakker [1980] to facilitate comparisons. We also recommend Apt [1981] as an excellent survey of material directly relevant to the subject of our monograph.

0.6.2 Organization

The four chapters of the book are divided into sections and subsections after the fashion of this Introduction. Thus, for example, the sixth section of Chapter 2 is referred to as "Section 2.6", and its tenth subsection as "subsection 2.6.10" or simply as "2.6.10".

0.6.3 References

The references of the book follow the standard form used in this Introduction, with one exception: because of the frequency of reference to De Bakker [1980], that book will be denoted [dB].

Chapter 1

Straight-line Programs

1.1 PRELIMINARIES: SIGNATURES AND STRUCTURES

1.1.1 Signatures

A signature may be defined as a pair $\Sigma=(\boldsymbol{Sort}, \boldsymbol{Func})$, where

(1) $\boldsymbol{Sort}=\boldsymbol{Sort}(\Sigma)$ is the finite set of sorts of Σ: the r *algebraic sorts* $1,\ldots,r$ (for some $r\geq 0$), the *numerical sort* N and the *boolean sort* B.

(2) $\boldsymbol{Func}=\boldsymbol{Func}(\Sigma)$ is a finite set of pairs (F,τ), where F is a *function symbol* and τ is the *type* of F, i.e. a tuple of the form $(m; i_1,\ldots,i_m,i)$ with $m\geq 0$, $i_j\in \boldsymbol{Sort}$ for $j=1,\ldots,m$ and $i\in \boldsymbol{Sort}$. F is called an m-ary function symbol, with *argument sorts* $(i_1,\ldots,i_m)$ and *value sort* i. In particular:

(a) if $i=$ B then F is a boolean-valued function symbol or *relation symbol* of type $(m; i_1,\ldots,i_m,\mathsf{B})$, and we may emphasize this by writing 'R' instead of 'F';

(b) if $m=0$, so that $\tau=(0;i)$, then F is a 0-ary function symbol or *individual constant symbol* of sort i.

(3) We assume that $\boldsymbol{Func}$ includes symbols for certain *standard functions* associated with sorts N and B:

(a) *arithmetical* function symbols for N, representing the following operations on the natural numbers: zero, successor, predecessor (where the predecessor of 0 is taken to be 0), addition and multiplication, and the order relation;

(b) *logical* function or relation symbols for B: the constant symbol **true**, and symbols for a complete set of propositional connectives, say **not** and **and**;

(*c*) *equality* symbols eq_{N} and eq_{B} for equality on sorts N and B respectively.

All function and relation symbols other than those in (*a*)—(*c*) above will be called *algebraic*.

As a loose terminology, we will write "F in ***Func***" to mean $(F,\tau)\in$ ***Func*** for some τ.

1.1.2 Remark on the inclusion of sorts N and B in the signature

Terms of sort N will be used as indices or "subscripts" for the array variables. However the main reason for including the sort N of the natural numbers $\{0,1,2,\ldots\}$ and the arithmetical functions is in connection with the assertion language, to ensure expressibility of pre- and postconditions for 'while' programs (Chapter 2) and recursive programs (Chapter 3).

We could instead have used the sort of the integers $\{\ldots,-2,-1,0,1,2,\ldots\}$ (as in [dB]), which, in practice, is more usual for array subscripts. However the sort of natural numbers is theoretically more satisfactory to handle, when considering definability and computability over our structures (in Chapters 3 and 4).

From now on, by "number" we will mean natural number.

By contrast, the inclusion of the sort B and the logical functions in the signature has no mathematical significance; it is for notational convenience, to ensure uniformity of treatment between booleans and program terms (expressions) of other sorts. It is also in accordance with programming practice.

1.1.3 Structures A of signature Σ

Now given a signature $\Sigma=(\boldsymbol{Sort},\boldsymbol{Func})$ where $\boldsymbol{Sort}=\{1,\ldots,r\}\cup\{\mathsf{N},\mathsf{B}\}$ and $\boldsymbol{Func}=\{(F_1,\tau_1),\ldots,(F_s,\tau_s)\}$, a *structure of signature* Σ, or Σ-*structure*, has the form

$$A=((A_i)_{i\in\boldsymbol{Sort}},(F_j^A)_{1\le j\le s})$$

where $A_1,\ldots,A_r$ are non-empty sets called the *algebraic domains*; $A_{\mathsf{N}}=\mathbb{N}=\{0,1,2,\ldots\}$, the domain of *natural numbers*; $A_{\mathsf{B}}=\mathbb{B}=\{\mathrm{tt},\mathrm{ff}\}$, the domain of *truth values*; and for $j=1,\ldots,s$, if $\tau_j=(m;i_1,\ldots,i_m,i)$ then

$$F_j^A\colon\ A_{i_1}\times\cdots\times A_{i_m}\to A_i.$$

Finally, the standard function symbols (see 1.1.1(3)) have their standard interpretations on $\mathbb{N}$ and $\mathbb{B}$. Thus, in particular, $\mathsf{true}^A=\mathrm{tt}$, the connectives **not** and **and** have their standard truth-functional interpretation, and eq_{N} and eq_{B} are interpreted as identity on A_{N} and A_{B} respectively.

Structures such as these, which contain the "standard domains" $\mathbb{N}$ and $\mathbb{B}$ for the distinguished sorts **N** and **B**, with the standard operations on them, will be called *standard structures.*

1.1.4 Classes of structures

We will consider computation by programs over a class $\mathbb{K}$ of Σ-structures. The class $\mathbb{K}$ is intended to model some class of implementations of a data type specification, as explained in Section 0.4. Our concern with computations which are uniform over $\mathbb{K}$ leads to a useful generalization of the mathematical theory of program correctness.

All classes $\mathbb{K}$ considered in this monograph satisfy one condition: *each structure in $\mathbb{K}$ is a standard structure* in the sense of 1.1.3. Such classes may be called *standard classes.* Henceforth, let $\mathbb{K}$ be any standard class of signature Σ.

1.1.5 The unspecified value ω; structures A^{ω} of signature Σ^{ω}

Given a structure

$$A = ((A_i)_{i \in \mathit{Sort}}, (F_j^A)_{1 \le j \le s})$$

let ω be a new object or symbol, representing an "unspecified value". For each sort i, let

$$A_i^{\omega} = A_i \cup \{\omega\}.$$

In particular, taking $i = \mathbf{B}$, we have

$$\mathbb{B}^{\omega} = A_{\mathbf{B}}^{\omega} = \{\mathbb{t}, \mathbb{f}, \omega\},$$

the set of truth values, including the "unspecified truth value" ω. This adjunction of ω will lead to a 3-valued logic for booleans, as we will see.

For each F in $\mathit{Func}(\Sigma)$ of type $(m; i_1, \ldots, i_m, i)$, we have a new interpretation of the function symbol F, namely

$$F^{A,\omega}: A_{i_1}^{\omega} \times \cdots \times A_{i_m}^{\omega} \to A_i^{\omega}$$

which *extends* F^A, by stipulating that the value is ω whenever any argument is ω.

Then A^{ω} is the structure

$$((A_i^{\omega})_{i \in \mathit{Sort}}, (F_j^{A,\omega})_{1 \le j \le s}, (\omega_i)_{i \in \mathit{Sort}})$$

where for each sort i, ω_i is ω, considered as an element of A_i^{ω}. The structure A^{ω} has signature Σ^{ω}, with function symbols F_j^{ω} for $j = 1, \ldots, s$, and $\mathbf{unspec}_i$ (denoting ω_i) for each sort i.

Finally, $\mathbb{K}^{\mathbb{u}}$ is the class of structures $A^{\mathbb{u}}$ for $A \in \mathbb{K}$.

1.1.6 Three-valued logic in $A^{\mathbb{u}}$

The definition of $F^{A,\mathbb{u}}$ in the last section, applied to the cases of the logical functions $F \equiv \mathbf{not}$ and $F \equiv \mathbf{and}$, gives us 3-valued truth functions, which extends the familiar 2-valued functions by stipulating a value of $\mathbb{u}$ whenever any argument is $\mathbb{u}$. This gives the *weak 3-valued logic* discussed in Kleene [1952], §64.

*1.1.7 Coding of $A^{\mathbb{u}}$ within A

The structure $A^{\mathbb{u}}$ can actually be represented or coded within A, so that our addition of $\mathbb{u}$, while of semantic interest, is mathematically inessential. This coding is accomplished by the device of representing an element y of $A_i^{\mathbb{u}}$ by a *pair* $(b,x) \in \mathbb{B} \times A_i$, where
— if $y \neq \mathbb{u}_i$ then $b = \mathbb{t}$ and $x = y$, and
— if $y = \mathbb{u}_i$ then $b = \mathbb{f}$ and x is some element of A_i.
We call the component b of the pair (b,x) a "flag" of the coding.

Under this representation, moreover, a function $F^{A,\mathbb{u}}$ of m arguments (with F in $\mathit{Func}(\Sigma)$) can be correspondingly represented in A by the pair of functions $(\mathbf{and}_m, F^A)$, where $\mathbf{and}_m : \mathbb{B}^m \to \mathbb{B}$ is the m-fold conjunction operation:

$$\mathbf{and}_m(b_1, \ldots, b_m) = \begin{cases} \mathbb{t} & \text{if } b_1 = b_2 = \cdots = b_m = \mathbb{t} \\ \mathbb{f} & \text{otherwise} \end{cases}$$

which is "inductively definable" over A.

The exact definition of "inductive definability" over a structure will be given in Chapter 4, and the precise nature of the "corresponding representation" of functions in $A^{\mathbb{u}}$ by functions in A is given in the Theorem in 4.1.12.

Another example of this kind of coding (involving flags) will be given in 3.5.3.

1.2 THE PROGRAMMING LANGUAGE

1.2.1 Syntax

The programming language $\mathit{ProgLang}_{sa} = \mathit{ProgLang}_{sa}(\Sigma)$ for the signature Σ ('*sa*' for "straight-line with arrays") has the following syntactic classes for each sort i of Σ:

(1) $\boldsymbol{SimVar}_i$, the class of *simple program variables* of sort i, denoted $v^i, w^i, \ldots$

(2) $\boldsymbol{ArrVar}_i$, the class of *array variables* of sort i, denoted $a^i, \ldots$

(3) $\boldsymbol{ProgVar}_i$, the class of *program variables*, i.e. the simple and array variables, of sort i:

$$\boldsymbol{ProgVar}_i =_{df} \boldsymbol{SimVar}_i \cup \boldsymbol{ArrVar}_i .$$

(There is no special notation for variables of this class.)

(4) $\boldsymbol{InterVar}_i$, the class of *intermediate variables* of sort i, denoted $V^i, \ldots$, and defined by

$$\boldsymbol{InterVar}_i =_{df} \boldsymbol{SimVar}_i \cup (\boldsymbol{ArrVar}_i \times \mathbb{N}),$$

in other words (using a variant of BNF notation, see [dB], p.18):

$$V^i ::= v^i \mid \langle a^i, n \rangle$$

where $n \in \mathbb{N}$.

This class will form the domain of the *proper states* (see 1.2.4).

(5) $\boldsymbol{SubVar}_i$, the class of *subscripted variables* of sort i, made by applying an array variable a^i of sort i to a program term t^{N} of sort N (see below) to form $a^i[t^{\mathsf{N}}]$.

(6) $\boldsymbol{ProgTerm}_i$, the class of *program terms* (or *expressions*) of sort i, denoted $t^i, \ldots$, and defined by:

$$t^i ::= v^i \mid a^i[t^{\mathsf{N}}] \mid F(t_1^{i_1}, \ldots, t_m^{i_m}) \mid \mathsf{if}\ t^{\mathsf{B}}\ \mathsf{then}\ t_1^i\ \mathsf{else}\ t_2^i\ \mathsf{fi}$$

where F is a function symbol in Σ of type $(m; i_1, \ldots, i_m, i)$.

In particular, for $i = \mathsf{B}$, we get the class of *boolean program terms* or *program booleans*,

$$\boldsymbol{ProgBool} =_{df} \boldsymbol{ProgTerm}_{\mathsf{B}},$$

denoted either $t^{\mathsf{B}}, \ldots$ (as above) or $b, \ldots$ This class is given by:

$$\begin{aligned} b ::= \quad & v^{\mathsf{B}} \mid a^{\mathsf{B}}[t^{\mathsf{N}}] \mid R(t_1^{i_1}, \ldots, t_m^{i_m}) \mid \mathsf{eq}_i(t_1^i, t_2^i) \mid \mathsf{true} \\ & \mid \mathsf{not}(b) \mid \mathsf{and}(b_1, b_2) \mid \mathsf{if}\ b\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2\ \mathsf{fi} \end{aligned}$$

where R is an algebraic relation symbol in Σ of type $(m; i_1, \ldots, i_m)$, and $i = \mathsf{N}$ or B.

We use the prefix notation for the equality relation and the boolean connectives, to distinguish these from the equality relation and connectives of the assertion language, which have different (2-valued!) truth conditions.

Further boolean expressions can be defined in a standard way:

$$\mathsf{false} \equiv_{df} \mathsf{not}(\mathsf{true})$$

$$\mathsf{or}(b_1, b_2) \equiv_{df} \mathsf{not}(\mathsf{and}(\mathsf{not}(b_1), \mathsf{not}(b_2)))$$

$$\mathsf{imply}(b_1, b_2) \equiv_{df} \mathsf{not}(\mathsf{and}(b_1, \mathsf{not}(b_2)))$$

$$\mathsf{if}\ b_1\ \mathsf{then}\ b_2\ \mathsf{fi} \equiv_{df} \mathsf{if}\ b_1\ \mathsf{then}\ b_2\ \mathsf{else}\ \mathsf{true}\ \mathsf{fi}$$

Finally, we have:

(7) $\boldsymbol{Statemt}_{sa} = \boldsymbol{Statemt}_{sa}(\Sigma)$, the class of (*straight-line, array*) *statements*, denoted $S, \ldots$, and defined by:

$$S ::= \mathsf{skip} \mid v^i := t^i \mid a^i[t^{\mathsf{N}}] := t^i \mid S_1; S_2 \mid \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}.$$

Thus a statement is either a 'skip', an *assignment* (for a simple or a subscripted variable), a *composition* of two statements or a *conditional* statement.

We need the 'skip' statement, since (as we will see later) this is not equivalent to $v := v$.

As usual, we let the statement 'if b then S fi' abbreviate 'if b then S else skip fi'.

We assume that each of the five classes of variables discussed above is countable. This is necessary for the sake of Gödel numberings that will be needed in later chapters. This assumption applies also to all classes of variables to be introduced later.

1.2.2 Further definitions and notational conventions

(1) We will try to adhere to the convention of [dB] and use *only the letters indicated* ('v' for simple variables, 'a' for array variables, 'S' for statements, etc.), possibly adorned with superscripts and subscripts, to denote arbitrary objects of the relevant syntactic classes.

(2) We will often drop the sort superscript (or subscript) i.

(3) We use the notation

$$\boldsymbol{SimVar}(\Sigma) =_{df} \bigcup_{i \in \boldsymbol{Sort}} \boldsymbol{SimVar}_i$$

$$\boldsymbol{ArrVar}(\Sigma) =_{df} \bigcup_{i \in \boldsymbol{Sort}} \boldsymbol{ArrVar}_i$$

etc. (and often drop the '(Σ)').

(4) For a syntactic expression E of any of the classes considered above (e.g. a program term or statement) we define:

$$\begin{aligned}\textbf{\textit{SimVar}}(E) &= \text{the set of simple variables occurring in } E,\\ \textbf{\textit{ArrVar}}(E) &= \text{the set of array variables occurring in } E,\\ \textbf{\textit{ProgVar}}(E) &= \text{the set of program variables occurring in } E\\ &= \textbf{\textit{SimVar}}(E)\cup\textbf{\textit{ArrVar}}(E).\end{aligned}$$

(5) For an expression E, $\textbf{\textit{compl}}(E)$ is the *structural complexity* of E.

There are many suitable definitions of $\textbf{\textit{compl}}(E)$, depending on the syntactic class of E. Thus, for example, for a program term $t \equiv F(t_1,\ldots,t_m)$, $\textbf{\textit{compl}}(t)$ can be defined as $\max_i(\textbf{\textit{compl}}(t_i))+1$. This definition, which gives $\textbf{\textit{compl}}(t)$ as the length of the maximum branch of the parse tree of t, is particularly appropriate for inductive definitions over $\textbf{\textit{ProgTerm}}$. Another possible definition of $\textbf{\textit{compl}}(E)$, which would in fact be satisfactory for our purposes, is simply the length of of E as a string of symbols.

(6) We use '$\equiv$' to denote syntactic identity between two expressions.

(7) For $V\in\textbf{\textit{InterVar}}$ and $M\subseteq\textbf{\textit{ProgVar}}$, '$V$ in M' means *either* $V\equiv v$ for some simple variable $v\in M$, *or* $V\equiv\langle a,n\rangle$ for some array variable $a\in M$ and $n\in\mathbb{N}$.

1.2.3 The programming language without arrays

We could also consider the simpler language $\textbf{\textit{ProgLang}}_s(\Sigma)$, i.e. the language without arrays. All the concepts to be introduced in this chapter could be modified in an obvious way to apply to this language.

1.2.4 Proper states

Let A be a structure in the class $\mathbb{K}$.

DEFINITIONS. (1) A *proper state* over A is a function from the intermediate variables of each sort i to A_i^{μ} (see 1.1.5), which assumes the unspecified value μ almost everywhere. In other words, it is a function of the form

$$\sigma = \bigcup_{i\in\textbf{\textit{Sort}}}\sigma_i$$

where $$\sigma_i : \textbf{\textit{InterVar}}_i \to A_i^{\mu}$$

for each sort i, such that $\sigma(V)\neq\mu$ for only finitely many intermediate variables V. (We say "proper state" to exclude the error state to be introduced later.)

For notational convenience, we will write '$\sigma(a,n)$' for '$\sigma(\langle a,n\rangle)$'.

(2) PR.STATE(A) is the set of proper states over A.

(3) The *domain* of a proper state σ is the set

$$dom(\sigma) =_{df} \{V \mid \sigma(V) \neq \mathbb{u}\}$$

(which is, by definition, finite).

1.2.5 Semantics of program terms

We will define, for each $A \in \mathbb{K}$, an *evaluation function* $\mathcal{R}_A$ for program terms, which is the union of a family $(\mathcal{R}_A^i)_{i \in Sort}$ of functions, where for each sort i

$$\mathcal{R}_A^i : \mathit{ProgTerm}_i \to (\mathrm{PR.STATE}(A) \to A_i^{\mathbb{u}}).$$

Thus for any program term t of sort i and proper state σ, $\mathcal{R}_A(t)(\sigma)$ will be an element of $A_i^{\mathbb{u}} = A_i \cup \{\mathbb{u}\}$. In particular, for a program boolean b, $\mathcal{R}_A(b)(\sigma)$ will be an element of $\mathbb{B}^{\mathbb{u}} = \{\mathbb{t}, \mathbb{f}, \mathbb{u}\}$.

The definition of $\mathcal{R}_A(t)(\sigma)$ is by induction on $\mathit{compl}(t)$:

$$\mathcal{R}_A(v^i)(\sigma) = \sigma(v^i)$$

$$\mathcal{R}_A(a^i[t])(\sigma) = \begin{cases} \sigma(a^i, \mathcal{R}_A(t)(\sigma)) & \text{if } \mathcal{R}_A(t)(\sigma) \neq \mathbb{u} \\ \mathbb{u} & \text{otherwise} \end{cases}$$

$$\mathcal{R}_A(F(t_1, \ldots, t_m))(\sigma) = F^{A,\mathbb{u}}(\mathcal{R}_A(t_1)(\sigma), \ldots, \mathcal{R}_A(t_m)(\sigma))$$

$$\mathcal{R}_A(\textsf{if } b \textsf{ then } t_1 \textsf{ else } t_2 \textsf{ fi})(\sigma) = \begin{cases} \mathcal{R}_A(t_1)(\sigma) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{t} \\ \mathcal{R}_A(t_2)(\sigma) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{f} \\ \mathbb{u} & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{u}. \end{cases}$$

Notice that we have avoided defining a "left value" $\mathcal{L}_A$ (as in [dB], Definition 4.2).

1.2.6 Error semantics: comparison with strong three-valued logic

Recall the definition of $F^{A,\mathbb{u}}$ in 1.1.5: this implies that at a given state, $F(t_1, \ldots, t_m)$ has the "unspecified value" $\mathbb{u}$ whenever any of $t_1, \ldots, t_m$ has this value. This is in accordance with the idea of *errors propagating* due to unspecified data. In particular, considering the case that F is a logical function, this means that a boolean of the form $\textsf{and}(b_1, b_2)$, $\textsf{or}(b_1, b_2)$, or $\textsf{imply}(b_1, b_2)$ has the value $\mathbb{u}$ whenever either b_1 or b_2 has. This is the *weak 3-valued logic* discussed in 1.1.6.

Thus the boolean $\textsf{imply}(b_1, b_2)$ has different truth conditions from $\textsf{if } b_1 \textsf{ then } b_2 \textsf{ fi}$, which has the value $\mathbb{t}$ when b_1 is $\mathbb{f}$, even if b_2 is $\mathbb{u}$.

The following theorem says that the intermediate variables not in $\mathbf{lhs}(S)$ do not change in value with the execution of S. It may be compared to [dB], Lemma 2.37.

THEOREM. *Suppose* $\sigma \neq \varepsilon$ *and* $\mathcal{M}_A(S)(\sigma) = \sigma' \neq \varepsilon$. *If* V *is not in* $\mathbf{lhs}(S)$ (*i.e.* $V \equiv v \notin \mathbf{lhs}(S)$ *or* $V \equiv \langle a, n \rangle$ *with* $a \notin \mathbf{lhs}(S)$), *then* $\sigma'(V) = \sigma(V)$.

PROOF. Induction on $\mathbf{compl}(S)$. □

*1.2.15 States which are specified on all relevant variables

The following propositions (or rather, their generalizations to later languages) will be used in Chapter 4 (4.4.2 via 2.2.10, and 4.4.4).

PROPOSITION 1. *Let* t *and* $\sigma \neq \varepsilon$ *be such that* $\sigma(v) \neq \mathbb{u}$ *for all* $v \in \mathbf{SimVar}(t)$, *and* $\mathbf{ArrVar}(t) = \emptyset$. *Then* $\mathcal{R}_A(t)(\sigma) \neq \mathbb{u}$.

PROOF. Induction on $\mathbf{compl}(t)$. □

PROPOSITION 2. *Let* S *and* $\sigma \neq \varepsilon$ *be such that* $\sigma(v) \neq \mathbb{u}$ *for all* $v \in \mathbf{SimVar}(S)$, *and* $\mathbf{ArrVar}(S) = \emptyset$. *Then* $\mathcal{M}_A(S)(\sigma) \neq \varepsilon$, *and, moreover,* $\mathcal{M}_A(S)(\sigma)(v) \neq \mathbb{u}$ *for all* $v \in \mathbf{SimVar}(S)$.

PROOF. Induction on $\mathbf{compl}(S)$. □

1.2.16 Isomorphism between structures; semantics abstraction theorem

Given two Σ-structures A and B, we define a Σ-*isomorphism* $\varphi: A \to B$ between them to be a family $\langle \varphi_i \mid i \in \mathbf{Sort}(\Sigma) \rangle$ of bijections $\varphi_i: A_i \to B_i$ between domains of A and B of each sort i of Σ, such that for each function symbol F of Σ, if F has type $(m; i_1, \ldots, i_m, i)$, then for all $\boldsymbol{x}_1 \in A_{i_1}, \ldots, \boldsymbol{x}_m \in A_{i_m}$,

$$\varphi_i(F^A(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)) = F^B(\varphi_{i_1}(\boldsymbol{x}_1), \ldots, \varphi_{i_m}(\boldsymbol{x}_m)).$$

In particular (taking $m = 0$), for any constant c of sort i, $\varphi_i(c^A) = c^B$.

It is easy to check that $\varphi_{\mathbb{N}}$ and $\varphi_{\mathbb{B}}$ are the identity functions on $\mathbb{N}$ and $\mathbb{B}$ respectively.

Clearly, any such Σ-isomorphism extends in a natural way to a $\Sigma^{\mathbb{u}}$-isomorphism $\varphi^{\mathbb{u}}: A^{\mathbb{u}} \to B^{\mathbb{u}}$, by stipulating $\varphi_i^{\mathbb{u}}(\mathbb{u}_i) = \mathbb{u}_i$.

More interestingly, any such Σ-isomorphism $\varphi: A \to B$ induces a bijection between the corresponding proper state spaces

$$\hat{\varphi}: \text{PR.STATE}(A) \to \text{PR.STATE}(B)$$

defined by

$$\hat{\varphi}(\sigma)(V^i) = \varphi_i^{\mathbb{u}}(\sigma(V^i)),$$

which can be extended to a mapping between the full state spaces

$$\hat{\varphi}_\varepsilon\colon \mathrm{STATE}(A) \to \mathrm{STATE}(B)$$

with $\hat{\varphi}_\varepsilon(\varepsilon) = \varepsilon$.

THEOREM. *Given a Σ-isomorphism $\varphi\colon A \to B$ and any $S \in \mathbf{Statemt}(\Sigma)$ (and with the notation as above), the following diagram commutes:*

$$\begin{array}{ccc} \mathrm{PR.STATE}(A) & \xrightarrow{\mathfrak{M}_A(S)} & \mathrm{STATE}(A) \\ \Big\downarrow \hat{\varphi} & & \Big\downarrow \hat{\varphi}_\varepsilon \\ \mathrm{PR.STATE}(B) & \xrightarrow{\mathfrak{M}_B(S)} & \mathrm{STATE}(B) \end{array}$$

PROOF. Induction on $\mathbf{compl}(S)$. □

In other words, the semantics of our programming language $\mathbf{ProgLang}_{sa}(\Sigma)$ satisfies the Program Semantics Abstraction Principle, as discussed in the Introduction (0.3.2).

1.3 ASSERTIONS

1.3.1 Syntax of the assertion language

For any signature Σ, let $\mathbf{Lang}_1(\Sigma)$ be the *first-order language* over Σ. It thus includes quantification over variables of all sorts in Σ.

Now the assertion language $\mathbf{AssLang}_{sa} = \mathbf{AssLang}_{sa}(\Sigma)$ ('*sa*' for 'straight-line with arrays') is based on $\mathbf{Lang}_1(\Sigma^u)$, the first-order language over Σ^u (see 1.1.5). More specifically, it will contain *assertion variables*, as well as program variables, to range over the domains of A^u, for $A \in \mathbb{K}$, but with quantification only over the assertion variables. The motivation for this is discussed in the next subsection.

The exact definition of the assertion language is as follows. We will define, in turn, the classes of *assertion variables*, *assertion terms* and *assertions*. Thus, for each sort i of Σ:

(1) $\mathbf{AssVar}_i$ is the class of *assertion variables* of sort i, denoted $x^i, y^i, z^i, \ldots$

(2) ***AssTerm***$_i$ is the class of *assertion terms* of sort i, denoted $s^i, \ldots$ It extends the class ***ProgTerm***$_i$ (1.2.1(6)) by the inclusion of assertion variables, and the constant unspec_i (denoting $\mathbb{u}_i$), thus:

$$s^i ::= x^i \mid v^i \mid a^i[s^{\mathsf{N}}] \mid F(s_1^{i_1}, \ldots, s_m^{i_m}) \mid \mathsf{if}\ s^{\mathsf{B}}\ \mathsf{then}\ s_1^i\ \mathsf{else}\ s_2^i\ \mathsf{fi} \mid \mathsf{unspec}_i$$

where F is in Σ, of type $(m; i_1, \ldots, i_m, i)$.

In particular, for $i = \mathsf{B}$, we get the class of *assertion booleans*

$$\textbf{\textit{AssBool}} =_{df} \textbf{\textit{AssTerm}}_{\mathsf{B}},$$

given by

$$\begin{aligned} s^{\mathsf{B}} ::= \quad & x^{\mathsf{B}} \mid v^{\mathsf{B}} \mid a^{\mathsf{B}}[s^{\mathsf{N}}] \mid R(s_1^{i_1}, \ldots, s_m^{i_m}) \mid \mathsf{eq}_i(s_1^i, s_2^i) \mid \mathsf{true} \mid \mathsf{not}(s^{\mathsf{B}}) \\ & \mid \mathsf{and}(s_1^{\mathsf{B}}, s_2^{\mathsf{B}}) \mid \mathsf{if}\ s^{\mathsf{B}}\ \mathsf{then}\ s_1^{\mathsf{B}}\ \mathsf{else}\ s_2^{\mathsf{B}}\ \mathsf{fi} \mid \mathsf{unspec}_{\mathsf{B}} \end{aligned}$$

where R is an algebraic relation symbol in Σ of type $(m; i_1, \ldots, i_m)$, and $i = \mathsf{N}$ or B.

(3) ***Assn*** $= \textbf{\textit{Assn}}(\Sigma)$ is the class of *assertions*, denoted $p, q, r, \ldots$

An *atomic assertion* has the form of an equality between two assertion terms of the same sort:

$$s_1^i = s_2^i,$$

where this equality symbol '=' is different from the relation symbol eq_i introduced already (when $i = \mathsf{N}$ or B), and has quite different semantics (as we will see below, 1.3.10).

Assertions are built up from atomic assertions by means of propositional connectives and quantification over assertion variables, thus:

$$p ::= s_1^i = s_2^i \mid \neg p \mid p_1 \wedge p_2 \mid \exists x^i[p]$$

Further logical operators can be defined in the standard way:

$$\begin{aligned} p_1 \vee p_2 &\equiv_{df} \neg(\neg p_1 \wedge \neg p_2) \\ p_1 \supset p_2 &\equiv_{df} \neg(p_1 \wedge \neg p_2) \\ p_1 \leftrightarrow p_2 &\equiv_{df} (p_1 \supset p_2) \wedge (p_2 \supset p_1) \\ \forall x^i[p] &\equiv_{df} \neg\exists x^i[\neg p]. \end{aligned}$$

1.3.2 Discussion

(1) *Program variables and assertion variables.*

This distinction between the two classes of variables can be understood as being due to the difference in the way the two classes arise.

The class of program variables, or rather the set of sorts which name their types, is determined *a priori* by the programming language under consideration. By contrast, the types of the (quantifiable) variables in the assertion language is determined *a posteriori* by the requirement that it be *expressible* for the programming language (see Section 1.6). In the context of the present chapter, the types of these two classes of variables is, in fact, the same (namely, one type corresponding to each sort of Σ), and so these classes could have been conflated here (as is generally done). However with the programming languages of the later chapters, the assertion language includes quantification over variables of other types (essentially, types of finite sequences over the sorts of Σ), as we will see, and so a distinction between these two classes of variables would become unavoidable at that point.

Another reason for making this syntactic distinction is that it reflects a clear conceptual distinction. Program variables of type i are *locations* in which values, namely elements of the domain A_i, can be stored. Assertion variables of type i "range over" (in the traditional logical terminology) the domain A_i^{U}, and, when bound by quantifiers within assertions, they give information about this domain.

It therefore seems methodologically and pedagogically sound to distinguish systematically between these two classes of variables, even in the present chapter.

(2) *Distinction between assertion booleans and assertions*

It is clear that $\boldsymbol{ProgTerm}_i \subset \boldsymbol{AssTerm}_i$ for each sort i, and in particular (taking $i = \mathsf{B}$) $\boldsymbol{ProgBool} \subset \boldsymbol{AssBool}$. However assertion booleans are *not* a particular type of assertion — in fact $\boldsymbol{AssBool}$ is disjoint from $\boldsymbol{Assn}$! — and hence $\boldsymbol{ProgBool}$ is also disjoint from $\boldsymbol{Assn}$. (This contrasts with the situation in [dB]: see Definitions 2.1(b) and 2.8 there.)

To recapitulate: program (or assertion) *booleans* are built up from program (or assertion) terms by the algebraic relations in Σ, and the logical functions of not, and and eq_i (for $i = \mathsf{N}$ and B). *Assertions* are built up from *atomic assertions* (which always have the form $s_1^i = s_2^i$, for some sort i) by the operations of $\neg$, $\wedge$ and $\exists$. Further, their semantics are quite different, as we shall see: for program or assertion booleans, it is 3-valued (1.3.7/8), while for assertions it is 2-valued (1.3.10).

Example. If R is a relation symbol in Σ, and $s_1, \ldots, s_m$ are program or assertion terms of suitable type, then '$R(s_1, \ldots, s_m)$' is a *boolean*, which, at a given state, may have one of three values: $\mathbb{t}$, $\mathbb{f}$ or $\mathbb{u}$. However '$R(s_1, \ldots, s_m) = \mathsf{true}$' is an *atomic assertion*, which must evaluate to either $\mathbb{t}$ or $\mathbb{f}$.

1.3.3 Further definitions

For any syntactic expression E, we define see 1.2.2(4)):

(1) $\boldsymbol{AssVar}(E)$ is the set of *free* assertion variables (i.e not bound by $\exists$) occurring in E.

(2) $\boldsymbol{Var}(E)$ is the set of *free* assertion and program variables occurring in E, i.e.

$$\boldsymbol{Var}(E) =_{df} \boldsymbol{AssVar}(E) \cup \boldsymbol{ProgVar}(E).$$

1.3.4 Notation convention for values of assertion variables

We use the following convention relating assertion variables to their possible values: the symbols $x^i, y^i, z^i, \ldots$ (or simply $x, y, z, \ldots$) denote assertion variables of sort i, while the corresponding boldface symbols $\boldsymbol{x}^i, \boldsymbol{y}^i, \boldsymbol{z}^i, \ldots$ (or $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \ldots$) denote elements of the domain A_i^{μ} in some structure A^{μ} of $\mathbb{K}^{\mu}$.

1.3.5 Valuations

In order to define the semantics of assertions, *states* alone are insufficient, since we also need assignments of meanings to the assertion variables.

DEFINITIONS. (1) A *valuation* (over A) is a function of the form

$$\rho = \bigcup_{i \in \boldsymbol{Sort}} \rho_i$$

where

$$\rho_i : \boldsymbol{AssVar}_i \to A_i^{\mu}$$

for each sort i.

(2) VAL(A) is the set of valuations over A, with elements denoted by $\rho, \ldots$

Note that we do not impose the condition with valuations (as for states, see 1.2.4) that $\rho(x) \neq \text{ш}$ for only finitely many x.

1.3.6 Variant of a valuation

Let $\rho \in \text{VAL}(A)$, $x^i \in \boldsymbol{AssVar}_i$ and $\boldsymbol{x} \in A_i^{\mu}$. Then $\rho\{\boldsymbol{x}/x^i\}$ is the valuation over A such that for all x'

$$\rho\{x/x^i\}(x') = \begin{cases} \rho(x') & \text{if } x' \not\equiv x \\ x & \text{if } x' \equiv x. \end{cases}$$

(Compare 1.2.9.)

1.3.7 Semantics of assertion terms

We will define, for each $A \in \mathbb{K}$, an *evaluation function* $\mathcal{S}_A$ for assertion terms, which is the union of a family $(\mathcal{S}_A^i)_{i \in \mathbf{Sort}}$ of functions, where for each sort i

$$\mathcal{S}_A^i : \mathbf{AssTerm}_i \to ((\mathrm{VAL}(A) \times \mathrm{PR.STATE}(A)) \to A_i^{\mu})$$

(compare 1.2.5). Thus for any assertion term s of sort i, valuation ρ over A and proper state σ over A, $\mathcal{S}_A(s)(\rho,\sigma)$ will be an element of A_i^{μ}.

The definition is by induction on $\mathbf{compl}(s)$, and is the obvious extension of the definition of $\mathcal{K}_A$ for program terms in 1.2.5, with the two new cases:

$$\mathcal{S}_A(x^i)(\rho, \sigma) = \rho(x^i)$$

$$\mathcal{S}_A(\mathsf{unspec}_i)(\rho, \sigma) = \mathbb{u}_i.$$

1.3.8 Consistency of semantics for program and assertion terms

From the definitions it is clear that for each sort i, $\mathbf{ProgTerm}_i \subset \mathbf{AssTerm}_i$ and their semantics (1.2.5, 1.3.7) are consistent, i.e. for any program term t and any ρ, σ:

$$\mathcal{S}_A(t)(\rho, \sigma) = \mathcal{K}_A(t)(\sigma).$$

In particular, considering terms of sort B, we have a *three-valued semantics* for assertion booleans, which extends that for program booleans.

1.3.9 Monotonicity for assertion terms

(Compare 1.2.8.) Suppose $\sigma, \sigma' \neq \varepsilon$.

THEOREM. *If* $\sigma \subsetneq \sigma'$ (rel $\mathbf{ProgVar}(s)$) *and* $\mathcal{S}_A(s)(\rho, \sigma) \neq \mathbb{u}$ *then* $\mathcal{S}_A(s)(\rho, \sigma) = \mathcal{S}_A(s)(\rho, \sigma')$.

PROOF. Induction on $\mathbf{compl}(s)$. □

COROLLARY. *If* $\sigma \simeq \sigma'$ (rel $\mathbf{ProgVar}(s)$) *then* $\mathcal{S}_A(s)(\rho, \sigma) = \mathcal{S}_A(s)(\rho, \sigma')$.

REMARK. In particular, if $\mathbf{ProgVar}(s) = \emptyset$, then $\mathcal{S}_A(s)(\rho, \sigma)$ does not depend on σ, in which case we may write it as $\mathcal{S}_A(s)(\rho, \cdot)$.

1.3.10 Semantics of assertions

We will define, for each $A \in \mathbb{K}$, an evaluation function for assertions:

$$\mathcal{I}_A\colon \ \textbf{\textit{Assn}} \to ((\mathrm{VAL}(A) \times \mathrm{PR.STATE}(A)) \to \mathbb{B})$$

(compare [dB], Definition 2.9). So for any assertion p, valuation ρ over A and proper state σ over A, $\mathcal{I}_A(p)(\rho, \sigma)$ will be either $\mathbb{t}$ or $\mathbb{f}$.

Thus assertions have the standard *two-valued semantics*, in contrast to (program or assertion) booleans (1.3.8).

The definition is, as usual, by induction on $\textbf{\textit{compl}}(p)$:

$$\mathcal{I}_A(s_1^i = s_2^i)(\rho, \sigma) = \begin{cases} \mathbb{t} & \text{if } \mathcal{S}_A(s_1^i)(\rho, \sigma) = \mathcal{S}_A(s_2^i)(\rho, \sigma) \\ \mathbb{f} & \text{otherwise.} \end{cases}$$

$\mathcal{I}_A(p_1 \wedge p_2)(\rho, \sigma)$ and $\mathcal{I}_A(\neg p)(\rho, \sigma)$ are defined according to the standard (two-valued) truth tables, and

$$\mathcal{I}_A(\exists x^i[p])(\rho, \sigma) = \begin{cases} \mathbb{t} & \text{if } \mathcal{I}_A(p)(\rho\{x/x^i\}) = \mathbb{t} \text{ for some } x \in A_i^{\mu} \\ \mathbb{f} & \text{otherwise.} \end{cases}$$

Notice again the difference between the semantics of the boolean $\mathbf{eq}_i(s_1, s_2)$ (for $i = \mathbf{N}$ or $\mathbf{B}$) and of the atomic assertion $s_1 = s_2$. If, at some ρ and σ, s_1 has the value $\mathbb{u}$, then so has $\mathbf{eq}_i(s_1, s_2)$, regardless of the value of s_2, but $s_1 = s_2$ will have the value either $\mathbb{t}$ or $\mathbb{f}$, depending on whether s_2 also has the value $\mathbb{u}$ or not.

1.3.11 Interpretations, satisfaction and validity

DEFINITIONS. (1) An *interpretation in* $\mathbb{K}$ is a triple $I = (A, \rho, \sigma)$ where $A \in \mathbb{K}$, $\rho \in \mathrm{VAL}(A)$ and $\sigma \in \mathrm{PR.STATE}(A)$.

(2) INTERP($\mathbb{K}$) is the class of interpretations in $\mathbb{K}$, with elements denoted by $I, \ldots$.

(3) If $I = (A, \rho, \sigma)$ and p is an assertion, then I *satisfies* p, written $I \models p$ or $A, \rho, \sigma \models p$, iff $\mathcal{I}_A(p)(\rho, \sigma) = \mathbb{t}$.

(4) The assertion p is $\mathbb{K}$-*valid*, written $\mathbb{K} \models p$, iff $I \models p$ for all $I \in$ INTERP($\mathbb{K}$).

1.3.12 Semantics of assertions with equivalent states and valuations

(Compare 1.3.9.) Suppose $\sigma, \sigma' \neq \varepsilon$.

THEOREM 1. *If* $\sigma \simeq \sigma'$ (rel $\textbf{\textit{ProgVar}}(p)$) *then* $\mathcal{I}_A(p)(\rho, \sigma) = \mathcal{I}_A(p)(\rho, \sigma')$.

PROOF. Induction on $\textbf{\textit{compl}}(p)$. □

REMARKS. (1) In particular, if $\mathbf{ProgVar}(p)=\emptyset$, then $\mathcal{I}_A(p)(\rho,\sigma)$ does not depend on σ, in which case we may write it as '$\mathcal{I}_A(p)(\rho,\cdot)$', and also write '$A,\rho \models p$' to mean $A,\rho,\sigma \models p$ for any $\sigma \neq \varepsilon$.

(2) There is no "monotonicity theorem" here, as there is with program terms (1.2.8), statements (1.2.13) and assertion terms (1.3.9).

DEFINITION (*Equivalence of valuations*). Let $M \subseteq \mathbf{AssVar}$. Then $\rho \simeq \rho'$ (rel M) iff for all $x \in M$, $\rho(x)=\rho'(x)$.

We now have the following analogue of Theorem 1 for equivalent valuations.

THEOREM 2. *If* $\rho \simeq \rho'$ (rel $\mathbf{AssVar}(p)$) *then* $\mathcal{I}_A(p)(\rho,\sigma)=\mathcal{I}_A(p)(\rho',\sigma)$.

PROOF. Similar to Theorem 1. □

1.3.13 Some simple facts on the semantics of program terms

PROPOSITION. *For any interpretation* $I=(A,\rho,\sigma)$ *and program term* t, *we have:*

(1) $\quad I \models t \neq \mathsf{unspec} \iff \mathcal{R}_A(t)(\sigma) \neq \mathbb{u}$.

Further, in the special case that $t \equiv b$, *a program boolean, we have:*

(2) $\quad I \models b = \mathsf{true} \iff \mathcal{R}_A(b)(\sigma) = \mathbb{t}$

(3) $\quad I \models b = \mathsf{false} \iff \mathcal{R}_A(b)(\sigma) = \mathbb{f}$

(4) $\quad I \models b \neq \mathsf{unspec} \iff \mathcal{R}_A(b)(\sigma) \neq \mathbb{u}$

$\qquad\qquad\qquad\qquad \iff \mathcal{R}_A(b)(\sigma) = \mathbb{t}$ or $\mathcal{R}_A(b)(\sigma) = \mathbb{f}$.

PROOF. Obvious, using 1.3.8. □

1.3.14 Substitution of a program term for a simple or subscripted variable

For use in defining the proof system (in particular the *assignment rules*: see 1.5.1) and in defining an assertion expressing the *weakest precondition* (see 1.6.5), we need to define the notions of the *substitution* of a program term t for a simple variable v or subscripted variable $a[t_0]$, of the same sort, in an assertion term or an assertion, denoted $E\langle t/v\rangle$ or $E\langle t/a[t_0]\rangle$ (where E is either an assertion term s or an assertion p). There are two cases:

(1) We first consider the case of substituting for a *simple variable*. The definitions of $s\langle t/v\rangle$ and $p\langle t/v\rangle$ are by induction, first on $\mathbf{compl}(s)$ and then on $\mathbf{compl}(p)$, and are completely straightforward, as in [dB], Definition 2.14, except that the case $p \equiv \exists x\,[p_1]$ is simpler here:

$$\exists x\,[p_1]\langle t/v\rangle \equiv \exists x\,[p_1\langle t/v\rangle],$$

since, by our separation of program and assertion variables, x cannot be identical to v, or contained in t.

(2) The case of substituting for a *subscripted variable* is similar, except for the subcase $s\langle t/a[t_0]\rangle$ with $s \equiv a[s_1]$ (as in [dB], Definition 4.4):

$$a[s_1]\langle t/a[t_0]\rangle \equiv \textsf{if } \mathsf{eq}_{\mathsf{N}}(s_1\langle t/a[t_0]\rangle, t_0) \textsf{ then } t \textsf{ else } a[s_1\langle t/a[t_0]\rangle] \textsf{ fi}$$

We now have (compare [dB], Lemma 2.16(c)):

THEOREM (Substitution for program variables). *Suppose* $\sigma \neq \varepsilon$.

(1) *If* $\bar{\sigma} = \sigma\{\mathcal{R}_A(t)(\sigma)/v\}$, *then*

(a) $\mathcal{S}_A(s\langle t/v\rangle)(\rho,\sigma) = \mathcal{S}_A(s)(\rho,\bar{\sigma})$,

(b) $A,\rho,\sigma \vDash p\langle t/v\rangle \iff A,\rho,\bar{\sigma} \vDash p$.

(2) *If* $\mathcal{R}_A(t_0)(\sigma) \neq \text{🕮}$ *and* $\tilde{\sigma} = \sigma\{\mathcal{R}_A(t)(\sigma)/\langle a, \mathcal{R}_A(t_0)(\sigma)\rangle\}$, *then*

(a) $\mathcal{S}_A(s\langle t/a[t_0]\rangle)(\rho,\sigma) = \mathcal{S}_A(s)(\rho,\tilde{\sigma})$,

(b) $A,\rho,\sigma \vDash p\langle t/a[t_0]\rangle \iff A,\rho,\tilde{\sigma} \vDash p$.

PROOF. In both parts, by induction, first on ***compl***(s) and then on ***compl***(p). □

Notice that we need *not* assume, in part (2) of the theorem, that $\mathcal{R}_A(t)(\sigma) \neq \text{🕮}$.

*1.3.15 Another type of substitution

We mention in passing another notion of substitution, which we need for the purpose of defining an assertion expressing the *strongest postcondition*. This is the substitution of an *assertion variable* for a simple or subscripted program variable of the same sort, where (in the latter case) the subscript is also an assertion variable:

$$p\langle x/v\rangle \quad \text{and} \quad p\langle x/a[z^{\mathsf{N}}]\rangle.$$

The definitions are similar to those in the preceding section (again, first defining such substitutions in assertion terms), except that one now has to rename bound variables in the case $p \equiv \exists y\,[p_1]$, to ensure that y is distinct from both x and z^{N}.

These substitutions satisfy substitution theorems similar to those in the preceding section, e.g., for $\sigma \neq \varepsilon$:

$$A, \rho, \sigma \models p\langle x/v\rangle \iff A, \rho, \sigma\{\rho(x)/v\} \models p$$
$$A, \rho, \sigma \models p\langle x/a[z]\rangle \iff A, \rho, \sigma\{\rho(x)/\langle a, \rho(z)\rangle\} \models p .$$

We do not emphasize this notion of substitution, because (*i*) the completeness theorem, at least in this chapter, does not depend on the expressibility of the strongest postcondition (see the Remark at the end of 1.6.7), and (*ii*) the expressibility of the strongest postcondition for the languages of Chapters 2 and 3 does not use this substitution.

*1.3.16 Substitution of one assertion variable for another

We must consider one more notion of substitution, for later use in Chapter 3 (3.3.4), namely, of one assertion variable for all free occurrences of another (of the same sort) in an assertion (with bound variables systematically renamed): $p\langle y/x\rangle$.

Again, such a substitution must first be defined in an assertion term, $s\langle y/x\rangle$, and then in an assertion, $p\langle y/x\rangle$, by induction on $\boldsymbol{compl}(s)$ and $\boldsymbol{compl}(p)$ respectively. We refrain from a complete definition, merely considering the case $p \equiv \exists z\,[p_1]$:

$$\exists z\,[p_1]\langle y/x\rangle \equiv \begin{cases} \exists z\,[p_1] & \text{if } z \equiv x \\ \exists z\,[p_1\langle y/x\rangle] & \text{if } z \not\equiv x,\ z \not\equiv y \\ \exists z_1[p_1\langle z_1/z\rangle\langle y/x\rangle] & \text{if } z \equiv y \not\equiv x \end{cases}$$

where z_1 is some assertion variable, of the same sort as z, not in $\boldsymbol{Var}(p_1) \cup \{x, y\}$ (say the first such one in some enumeration of $\boldsymbol{AssVar}$).

We will also need (in 3.3.4) the following

THEOREM (Substitution for assertion variables). *For* $\sigma \neq \varepsilon$,

(*a*) $\mathcal{S}_A(s\langle y/x\rangle)(\rho, \sigma) = \mathcal{S}_A(s)(\rho\{\rho(y)/x\}, \sigma)$,

(*b*) $\mathcal{T}_A(p\langle y/x\rangle)(\rho, \sigma) = \mathcal{T}_A(p)(\rho\{\rho(y)/x\}, \sigma)$.

PROOF. Induction on $\boldsymbol{compl}(s)$ and $\boldsymbol{compl}(p)$. □

*1.3.17 Congruent assertions

Finally, for use in the next chapter, (2.6.7) we mention here the syntactic notion of congruence of assertions. Two assertions, p and p', are called *congruent*, $p \cong p'$, if they differ only in the naming of bound variables. (We leave as an exercise an exact definition, by induction on $\boldsymbol{compl}(p)$.) Then we have:

PROPOSITION. *If* $p \cong p'$ *then* $\mathcal{T}_A(p) \cong \mathcal{T}_A(p')$.

1.4 CORRECTNESS FORMULAE

1.4.1 Syntax

The class $\boldsymbol{Form}_{sa} = \boldsymbol{Form}_{sa}(\Sigma)$ of *correctness formulae* (relative to Σ), denoted $f, \ldots$, is defined by:

$$f ::= \{p\}S\{q\} \mid p.$$

There are thus two types of correctness formulae:
(1) the "Floyd-Hoare formulae" or "specified programs" $\{p\}S\{q\}$, and
(2) assertions p, which can be thought of as assertions about the data.

1.4.2 Semantics

(Compare 1.3.11.)

DEFINITIONS. (1) (*Satisfaction.*) We will define, for an interpretation $I = (A, \rho, \sigma)$ and correctness formula f, the notion I *satifies* f, written $I \models f$ or $A, \rho, \sigma \models f$.

Case 1. $f \equiv \{p\}S\{q\}$. Then $A, \rho, \sigma \models f$ iff

$$A, \rho, \sigma \models p \Rightarrow (\mathcal{M}_A(S)(\sigma) \neq \varepsilon \text{ and } A, \rho, \mathcal{M}_A(S)(\sigma) \models q).$$

Case 2. $f \equiv p$. Then $I \models f$ as in 1.3.11.

(2) ($\mathbb{K}$-*validity.*) A correctness formula f is $\mathbb{K}$-*valid*, written $\mathbb{K} \models f$, iff $I \models f$ for all $I \in \mathrm{INTERP}(\mathbb{K})$.

1.5 A PROOF SYSTEM; SOUNDNESS

1.5.1 The proof system

We will define a proof system $\boldsymbol{ProofSys}_{sa} = \boldsymbol{ProofSys}_{sa}(\mathbb{K})$ for deriving $\mathbb{K}$-valid correctness formulae. We will prove soundness of the system below (1.5.2), and completeness later (1.7.1, after having proved expressibility of the weakest precondition).

The proof rules can be divided into three groups: (A) rules for the programming language constructs, (B) the $\mathbb{K}$-oracle axiom, and (C) "logical" rules. They are listed below.

(A) *Rules for the programming language constructs.*

(A.1) *The* 'skip' *rule*:

$$\frac{p \supset q}{\{p\}\mathsf{skip}\{q\}}$$

(A.2) *Assignment*:

(simple variable:) $$\frac{p \supset (t \neq \textsf{unspec} \wedge q\langle t/v\rangle)}{\{p\}v := t\{q\}}$$

(subscripted variable:)

$$\frac{p \supset (t_0 \neq \textsf{unspec} \wedge t \neq \textsf{unspec} \wedge q\langle t/a[t_0]\rangle)}{\{p\}a[t_0] := t\{q\}}$$

(A.3) *Composition*:

$$\frac{\{p\}S_1\{r\},\quad \{r\}S_2\{q\}}{\{p\}S_1;S_2\{q\}}$$

(A.4) *Conditional*:

$$\frac{\{p \wedge (b = \textsf{true})\}S_1\{q\},\quad \{p \wedge (b = \textsf{false})\}S_2\{q\},\quad p \supset (b \neq \textsf{unspec})}{\{p\}\textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \textsf{ fi}\{q\}}$$

(B) *The* $\mathbb{K}$*-oracle rule*:

Every $\mathbb{K}$-valid assertion is taken as an *axiom*.

(C) *Logical rules*.

(C.1) *Consequence*:

$$\frac{p \supset p_1,\quad \{p_1\}S\{q_1\},\quad q_1 \supset q}{\{p\}S\{q\}}$$

More rules will be added to groups (A) and (C) in Chapters 2 and 3, as the programming language is extended.

Note that the rules of group (A) are independent of $\mathbb{K}$. $\mathbb{K}$ enters the proof system via the $\mathbb{K}$-oracle rule, and affects the correctness formulae $\{p\}S\{q\}$ via the consequence rule.

DEFINITIONS. (1) The *axioms* of $\boldsymbol{ProofSys}_{sa}(\mathbb{K})$ are the $\mathbb{K}$-valid assertions, given by the oracle rule. The *inference rules* are the remaining proof rules.

(2) A *derivation* in $\boldsymbol{ProofSys}_{sa}(\mathbb{K})$ may be defined as a finite sequence of correctness formulae $D \equiv (f_1, \ldots, f_n)$, where, for $i = 1, \ldots, n$, f_i is either an axiom, or the conclusion of (an instance of) one of the inference rules, the premisses of which occur among $f_1, \ldots, f_{i-1}$. D is said to be a *derivation* of f_n. Its *length* is n.

(3) f is *derivable* in $\boldsymbol{ProofSys}_{sa}(\mathbb{K})$, written $\mathbb{K} \vdash f$, if there is a derivation in $\boldsymbol{ProofSys}_{sa}(\mathbb{K})$ of f.

1.5.2 Soundness

DEFINITION. An inference

$$\frac{f_1, \ldots, f_n}{f}$$

is $\mathbb{K}$-*valid* iff $\mathbb{K}$-validity of the premisses implies $\mathbb{K}$-validity of the conclusion, i.e.

$$(\mathbb{K} \models f_i \text{ for } i = 1, \ldots, n) \Rightarrow \mathbb{K} \models f.$$

THEOREM. *The system* $\textbf{\textit{ProofSys}}_{sa}(\mathbb{K})$ *is sound relative to* $\mathbb{K}$, *i.e. for any* $f \in \textbf{\textit{Form}}_{sa}(\Sigma)$,

$$\mathbb{K} \vdash f \Rightarrow \mathbb{K} \models f.$$

PROOF. We will show that each of the inference rules of $\textbf{\textit{ProofSys}}_{sa}(\mathbb{K})$ is $\mathbb{K}$-valid. The theorem then follows, by induction on the length of a derivation of f.

(A.1) The **'skip'** rule: this is clear.

(A.2) The assignment rules.
(a) For simple variables: assume

$$\mathbb{K} \models p \supset (t \neq \textbf{unspec} \wedge q\langle t/v \rangle). \tag{1}$$

We must show: $\mathbb{K} \models \{p\} v := t \{q\}$. So take an interpretation $I = (A, \rho, \sigma)$ in $\mathbb{K}$, such that

$$A, \rho, \sigma \models p \tag{2}$$

and put

$$\sigma' = \mathcal{M}_A(v := t)(\sigma). \tag{3}$$

We must show: $\sigma' \neq \varepsilon$ and $A, \rho, \sigma' \models q$. By (1) and (2),

$$A, \rho, \sigma \models t \neq \textbf{unspec} \tag{4}$$

and

$$A, \rho, \sigma \models q\langle t/v \rangle. \tag{5}$$

By (4) and 1.3.13,

$$\mathcal{R}_A(t)(\sigma) \neq \text{ш}. \tag{6}$$

By (3), (6) and the definition of $\mathcal{M}_A$ (1.2.11),

$$\sigma' = \sigma\{\mathcal{R}_A(t)(\sigma)/v\} \ (\neq \varepsilon). \tag{7}$$

Finally, by (5), (7) and Theorem 1(b) in 1.3.14, $A, \rho, \sigma' \models q$.
(b) For subscripted variables: Exercise.

(A.3) The composition rule. Suppose

$$\mathbb{K} \models \{p\} S_1 \{r\} \tag{8}$$

and

$$\mathbb{K} \models \{r\} S_2 \{q\}. \tag{9}$$

We must show: $\mathbb{K} \models \{p\} S_1; S_2 \{q\}$. So take an interpretation $I = (A, \rho, \sigma)$ in $\mathbb{K}$ such that

$$A, \rho, \sigma \models p \tag{10}$$

and put

$$\sigma' = \mathcal{M}_A(S_1; S_2)(\sigma). \tag{11}$$

We must show: $\sigma' \neq \varepsilon$ and $A, \rho, \sigma' \models q$. Let

$$\sigma'' = \mathcal{M}_A(S_1)(\sigma). \tag{12}$$

By (8) and (10),

$$\sigma'' \neq \varepsilon \tag{13}$$

and

$$A, \rho, \sigma'' \models r. \tag{14}$$

By (11), (12), (13) and the definition of $\mathcal{M}_A$,

$$\sigma' = \mathcal{M}_A(S_2)(\sigma''). \tag{15}$$

Finally, by (9), (14) and (15), $\sigma' \neq \varepsilon$ and $A, \rho, \sigma' \models q$.

(A.4) The conditional rule. Suppose

$$\mathbb{K} \models \{p \wedge (b = \textsf{true})\} S_1 \{q\}, \tag{16}$$

$$\mathbb{K} \models \{p \wedge (b = \textsf{false})\} S_2 \{q\} \tag{17}$$

and

$$\mathbb{K} \models p \supset (b \neq \textsf{unspec}). \tag{18}$$

We must show: $\mathbb{K} \models$ **if** b **then** S_1 **else** S_2 **fi**. So take an interpretation $I = (A, \rho, \sigma)$ such that

$$A, \rho, \sigma \models p \tag{19}$$

and put

$$\sigma' = \mathcal{M}_A(\textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \textsf{ fi})(\sigma). \tag{20}$$

We must show: $\sigma \neq \varepsilon$ and $A, \rho, \sigma' \models q$. By (18) and (19),

$$A, \rho, \sigma \models b \neq \textsf{unspec},$$

hence (by 1.3.13)

$$\mathcal{R}_A(b)(\sigma) = \mathbb{t} \text{ or } \mathbb{f}.$$

So there are two cases:

Case 1. $\mathcal{R}_A(b)(\sigma) = \mathbb{t}$. (21)

By (20) and the definition of $\mathcal{M}_A$,

$$\sigma' = \mathcal{M}_A(S_1)(\sigma). \tag{22}$$

By (19) and (21), $A, \rho, \sigma \models p \wedge (b = \mathbf{true})$,

so by (16) and (22), $\sigma' \neq \varepsilon$ and $A, \rho, \sigma' \models q$.

Case 2. $\mathcal{R}_A(b)(\sigma) = \mathbb{f}$. Similar, using (17).

(B.2) The consequence rule: Exercise. □

1.6 PREDICATES; STATE TRANSFORMERS; THE WEAKEST PRECONDITION AND STRONGEST POSTCONDITION

1.6.1 Predicates

DEFINITIONS. (1) A *state predicate* on a structure $A \in \mathbb{K}$ is a function

$$\pi\colon \text{PR.STATE}(A) \to \mathbb{B}.$$

(2) A state predicate π on A is *finitely based* if there exists a finite set $M \subset \textbf{\textit{ProgVar}}$ such that for all $\sigma, \sigma' \in \text{PR.STATE}(A)$,

$$\sigma \simeq \sigma' \text{ (rel } M) \Rightarrow \pi(\sigma) = \pi(\sigma')$$

(see Definition 2 in 1.2.12).

EXAMPLE. For any $p \in \textbf{\textit{Assn}}$ and $\rho \in \text{VAL}(A)$, let us use the notation

$$\mathcal{I}_A(p)(\rho) =_{df} \lambda\sigma \in \text{PR.STATE}(A).\mathcal{I}_A(p)(\rho, \sigma).$$

Then $\mathcal{I}_A(p)(\rho)$ is a finitely based predicate, by 1.3.12.

(3) A state predicate π on A is *expressible over A in the assertion language* $\textbf{\textit{AssLang}}_{sa}$ (relative to a valuation ρ) if there is an assertion p such that $\mathcal{I}_A(p)(\rho) = \pi$. In that case we say that p *expresses* π *over* A (rel ρ).

1.6.2 State transformers

DEFINITIONS. (1) A *state transformer* on A is a function

$$\Phi\colon \text{PR.STATE}(A) \to \text{STATE}(A)$$

which is *monotone*, in the sense that for $\sigma, \sigma' \in \text{PR.STATE}(A)$,

$$\sigma \subsetneq \sigma' \Rightarrow \Phi(\sigma) \subsetneq \Phi(\sigma') \tag{1}$$

(see Definition 1 in 1.2.12).

(2) A state transformer Φ on A is *finitely based* if there exists a finite set $M \subset \mathbf{ProgVar}$ such that (a) for all $\sigma,\sigma' \in \mathrm{PR.STATE}(A)$,

$$\sigma \simeq \sigma' \,(\mathrm{rel}\ M) \Rightarrow \Phi(\sigma) \simeq \Phi(\sigma') \,(\mathrm{rel}\ M), \tag{2a}$$

and (b) for all V *not* in M and all $\sigma \in \mathrm{PR.STATE}(A)$,

$$\Phi(\sigma)(V) = \sigma(V). \tag{2b}$$

REMARK. Conditions (2a) and (2b) above are called *aloofness* and *stability* respectively in Schwarz [1977]. Together they imply the following, which is weaker than these conditions together but stronger than (2a) alone (for $\sigma,\sigma' \in \mathrm{PR.STATE}(A)$ and M the finite set given in (2)):

$$\text{for all } M' \supseteq M,\ \sigma \simeq \sigma' \,(\mathrm{rel}\ M') \Rightarrow \Phi(\sigma) \simeq \Phi(\sigma') \,(\mathrm{rel}\ M'). \tag{2a'}$$

Also, under the finite base assumption (2), the monotonicity condition (1) is equivalent to each of the following (again, for $\sigma,\sigma' \in \mathrm{PR.STATE}(A)$ and M the finite set given in (2)):

$$\sigma \subsetneq \sigma' \,(\mathrm{rel}\ M) \Rightarrow \Phi(\sigma) \subsetneq \Phi(\sigma') \,(\mathrm{rel}\ M); \tag{1'}$$

$$\text{for all } M' \supseteq M,\ \sigma \subsetneq \sigma' \,(\mathrm{rel}\ M') \Rightarrow \Phi(\sigma) \subsetneq \Phi(\sigma') \,(\mathrm{rel}\ M'). \tag{1''}$$

PROPOSITION. *For any statement S, $\mathfrak{M}_A(S)$ is a finitely based state transformer.*

PROOF. By the theorems in 1.2.13 and 1.2.14. □

1.6.3 Weakest precondition and strongest postcondition

DEFINITIONS. (1) (*Weakest precondition.*) Given a state transformer Φ and predicate π, both on A, the *weakest precondition* of Φ and π, written $\mathrm{WP}_A(\Phi,\pi)$, is the predicate on A which holds at $\sigma \in \mathrm{PR.STATE}(A)$ (i.e. its value at σ is tt) iff

$$\Phi(\sigma) \neq \varepsilon \text{ and } \pi(\Phi(\sigma)) = \mathrm{tt}.$$

(2) (*Strongest postcondition.*) Given a state transformer Φ and predicate π, both on A, the *strongest postcondition* of Φ and π, written $\mathrm{SP}_A(\pi,\Phi)$, is the predicate on A which holds at $\sigma \in \mathrm{PR.STATE}(A)$ iff

$$\text{there exists } \sigma' \neq \varepsilon \text{ such that } \pi(\sigma') = \mathrm{tt} \text{ and } \Phi(\sigma') = \sigma.$$

PROPOSITION. *If Φ and π are finitely based, then so are* $\mathrm{WP}_A(\Phi,\pi)$ *and* $\mathrm{SP}_A(\pi,\Phi)$.

1.6.4 Expressibility of the weakest precondition and strongest postcondition: Definitions

Given a statement S and assertion p, we say that the weakest precondition (or strongest postcondition, respectively) of S and p is $\mathbb{K}$-*expressible* in the assertion language $\boldsymbol{AssLang}_{sa}$ if there is an assertion q of that language, such that for all $A \in \mathbb{K}$ and $\rho \in \mathrm{VAL}(A)$,

$$\mathcal{T}_A(q)(\rho) = \mathrm{WP}_A(\mathcal{M}_A(S), \mathcal{T}_A(p)(\rho))$$

$$(\text{or } \mathcal{T}_A(q)(\rho) = \mathrm{SP}_A(\mathcal{T}_A(p)(\rho), \mathcal{M}_A(S)), \text{ respectively}).$$

In this case we say that q *expresses* the weakest precondition (or strongest postcondition, respectively) of S and p, *uniformly over* $\mathbb{K}$.

We will prove the expressibility of the weakest precondition and strongest postcondition in the next two subsections.

We should note that Olderog [1983] has shown that for any finitely based state transformer, expressibility of that transformer, of its weakest precondition, and of its strongest postcondition, are all three equivalent to one another (for a given first-order language, over a fixed structure).

1.6.5 Expressibility of the weakest precondition

We will give an effective method for constructing, from a statement S and assertion p, an assertion denoted $\mathsf{wp}_{sa}[S, p]$ which we will show expresses the weakest precondition of S and p, uniformly over $\mathbb{K}$.

DEFINITION. The assertion $\mathsf{wp}_{sa}[S, p]$ is defined by induction on $\boldsymbol{compl}(S)$. (Below and elsewhere, we omit the subscript '*sa*'.)

$$\mathsf{wp}[\mathsf{skip}, p] \equiv p$$

$$\mathsf{wp}[v := t, p] \equiv (t \neq \mathsf{unspec}) \wedge p\langle t/v \rangle$$

$$\mathsf{wp}[a[t_0] := t, p] \equiv (t_0 \neq \mathsf{unspec}) \wedge (t \neq \mathsf{unspec}) \wedge p\langle t/a[t_0] \rangle$$

$$\mathsf{wp}[S_1; S_2, p] \equiv \mathsf{wp}[S_1, \mathsf{wp}[S_2, p]]$$

$$\mathsf{wp}[\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, p] \equiv ((b = \mathsf{true}) \wedge \mathsf{wp}[S_1, p]) \vee ((b = \mathsf{false}) \wedge \mathsf{wp}[S_2, p]).$$

THEOREM. $\mathsf{wp}[S,p]$ *expresses the weakest precondition of* S *and* p, *uniformly over* $\mathbb{K}$. *In other words, for all* $(A,\rho,\sigma)\in \mathrm{INTERP}(\mathbb{K})$, *if* $\sigma' = \mathcal{M}_A(S)(\sigma)$, *then*

$$A,\rho,\sigma \models \mathsf{wp}[S,p] \iff \sigma'\neq\varepsilon \text{ and } A,\rho,\sigma' \models p.$$

PROOF. By induction on $\mathit{compl}(S)$.

Case 1. $S \equiv \mathsf{skip}$. Clear.

Case 2. $S \equiv v := t$. Then

$$\mathsf{wp}[S,p] \equiv (t \neq \mathsf{unspec}) \wedge p\langle t/v\rangle. \tag{1}$$

Also (by the definition of $\mathcal{M}_A$)

$$\sigma' = \begin{cases} \sigma\{\mathcal{R}_A(t)(\sigma)/v\} \ (\neq\varepsilon) & \text{if } \mathcal{R}_A(t)(\sigma)\neq \text{ш} \\ \varepsilon & \text{if } \mathcal{R}_A(t)(\sigma) = \text{ш}. \end{cases} \tag{2}$$

Now $A,\rho,\sigma \models \mathsf{wp}[S,p]$

$\iff$ $A,\rho,\sigma \models (t\neq\mathsf{unspec})\wedge p\langle t/v\rangle$ (by (1))

$\iff$ $\mathcal{R}_A(t)(\sigma)\neq$ ш and $A,\rho,\sigma \models p\langle t/v\rangle$ (by 1.3.13)

$\iff$ $\mathcal{R}_A(t)(\sigma)\neq$ ш and $A,\rho,\sigma\{\mathcal{R}_A(t)(\sigma)/v\} \models p$ (by 1.3.14, Theorem 1(b))

$\iff$ $\sigma'\neq\varepsilon$ and $A,\rho,\sigma' \models p$ (by (2)).

Case 3. $S \equiv a[t_0] := t$. Similar. (Use 1.3.14, Theorem 2(b).)

Case 4. $S \equiv S_1;S_2$. Then

$$\mathsf{wp}[S,p] \equiv \mathsf{wp}[S_1,\mathsf{wp}[S_2,p]]. \tag{3}$$

Put $\sigma'' = \mathcal{M}_A(S_1)(\sigma)$. Then (by definition)

$$\sigma' = \begin{cases} \mathcal{M}_A(S_2)(\sigma'') & \text{if } \sigma''\neq\varepsilon \\ \varepsilon & \text{if } \sigma'' = \varepsilon. \end{cases} \tag{4}$$

Now $A,\rho,\sigma \models \mathsf{wp}[S,p]$

$\iff$ $A,\rho,\sigma \models \mathsf{wp}[S_1,\mathsf{wp}[S_2,p]]$ (by (3))

$\iff$ $\sigma''\neq\varepsilon$ and $A,\rho,\sigma'' \models \mathsf{wp}[S_2,p]$ (induction hypothesis)

$\iff$ $\sigma'\neq\varepsilon$ and $A,\rho,\sigma' \models p$ (induction hypothesis again and (4)).

Case 5. $S \equiv$ **if** b **then** S_1 **else** S_2 **fi**. Then

$$\mathbf{wp}[S,p] \equiv ((b = \mathbf{true}) \wedge \mathbf{wp}[S_1,p]) \vee ((b = \mathbf{false}) \wedge \mathbf{wp}[S_2,p]). \quad (5)$$

Also (by definition)

$$\sigma' = \begin{cases} \mathcal{M}_A(S_1)(\sigma) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{t} \\ \mathcal{M}_A(S_2)(\sigma) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{f} \\ \varepsilon & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{u}. \end{cases} \quad (6)$$

Now $A, \rho, \sigma \models \mathbf{wp}[S,p]$

$\Leftrightarrow$ $\begin{cases} \text{either } (\mathcal{R}_A(b)(\sigma) = \mathbb{t} \text{ and } A, \rho, \sigma \models \mathbf{wp}[S_1,p]) \\ \text{or} \quad (\mathcal{R}_A(b)(\sigma) = \mathbb{f} \text{ and } A, \rho, \sigma \models \mathbf{wp}[S_2,p]) \end{cases}$ (by (5) and 1.3.13)

$\Leftrightarrow$ $\begin{cases} \text{either } (\mathcal{R}_A(b)(\sigma) = \mathbb{t} \text{ and } \sigma' \neq \varepsilon \text{ and } A, \rho, \sigma' \models p) \\ \text{or} \quad (\mathcal{R}_A(b)(\sigma) = \mathbb{f} \text{ and } \sigma' \neq \varepsilon \text{ and } A, \rho, \sigma' \models p) \end{cases}$ (ind. hyp. and (6))

$\Leftrightarrow$ $\mathcal{R}_A(b)(\sigma) \neq \mathbb{u}$ and $\sigma' \neq \varepsilon$ and $A, \rho, \sigma' \models p$

$\Leftrightarrow$ $\sigma' \neq \varepsilon$ and $A, \rho, \sigma' \models p$ (by (6)). □

1.6.6 Properties of the weakest precondition

As simple consequences of the theorem in the last subsection, we have

COROLLARIES. (1) $\mathbb{K} \models \{q\}S\{p\} \Leftrightarrow \mathbb{K} \models q \supset \mathbf{wp}[S,p]$.

In particular, taking $q \equiv \mathbf{wp}[S,p]$:

$$\mathbb{K} \models \{\mathbf{wp}[S,p]\}S\{p\}.$$

(2) (Intermediate assertion.)

$$\mathbb{K} \models \{q\}S_1;S_2\{p\} \Leftrightarrow \mathbb{K} \models \{q\}S_1\{\mathbf{wp}[S_2,p]\}.$$

PROOF. Clear. The right-to-left implication in (2) follows from (1) and the validity of the composition rule. □

*1.6.7 Expressibility of the strongest postcondition

We will give an effective method for constructing, from a statement S and an assertion p, an assertion denoted $\mathbf{sp}_{sa}[p,S]$ which, we will show, expresses the strongest postcondition of S and p.

DEFINITION. The assertion $\mathbf{sp}_{sa}[p,S]$ is defined by induction on ***compl***(S). (Again we drop the subscript '*sa*'.)

$$\mathsf{sp}[p, \mathsf{skip}] \equiv p$$

$$\mathsf{sp}[p, v := t] \equiv \exists x\,[\,p\langle x/v\rangle \wedge v = t\langle x/v\rangle \neq \mathsf{unspec}\,] \quad (\text{where } x \notin \mathit{Var}(p,t))$$

$$\mathsf{sp}[p, a[t_0] := t] \equiv \exists x, z^{\mathbb{N}}[\; p\langle x/a[z^{\mathbb{N}}]\rangle \wedge y = t_0\langle x/a[z^{\mathbb{N}}]\rangle \neq \mathsf{unspec} \wedge a[z^{\mathbb{N}}] = t\langle x/a[z^{\mathbb{N}}]\rangle \neq \mathsf{unspec}\,] \quad (\text{where } x, z^{\mathbb{N}} \notin \mathit{Var}(p, t_0, t))$$

$$\mathsf{sp}[p, S_1; S_2] \equiv \mathsf{sp}[\mathsf{sp}[p, S_1], S_2]$$

$$\mathsf{sp}[p, \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}] \equiv \mathsf{sp}[p \wedge (b = \mathsf{true}), S_1] \vee \mathsf{sp}[p \wedge (b = \mathsf{false}), S_2].$$

Note that the substitutions '$\langle x/v\rangle$' and '$\langle x/a[z^{\mathbb{N}}]\rangle$' used in the above definition were discussed in 1.3.15.

THEOREM. $\mathsf{sp}[p, S]$ *expresses the strongest postcondition of* S *and* p, *uniformly over* $\mathbb{K}$. *In other words, for all* $(A, \rho, \sigma) \in \mathrm{INTERP}(\mathbb{K})$,

$$A, \rho, \sigma \models \mathsf{sp}[p, S] \iff \text{for some } \sigma' \neq \varepsilon,\ A, \rho, \sigma' \models p \text{ and } \mathcal{M}_A(S)(\sigma') = \sigma.$$

PROOF. By induction on $\mathit{compl}(S)$. For the case that S is an assignment, use the theorems on substitution stated in 1.3.15. We omit details. □

We remark that this theorem (and the corollaries in the next subsection) are not actually needed in the sequel, since for the proof of completeness of the proof system (1.7.1) we can use the expressibility of the weakest precondition instead of the strongest postcondition.

*1.6.8 Properties of the strongest postcondition

Now the analogues of the corollaries in 1.6.6 do not all hold. In particular, the correctness formula

$$\{p\}S\{\mathsf{sp}[p, S]\}$$

is *not* valid in general. For a counterexample, take $S \equiv v \neq v$ and $p \equiv \mathsf{true}$. Then $\mathsf{sp}[p, S]$ is (equivalent to) $v := \mathsf{unspec}$. But the formula $\{\mathsf{true}\}\, v := v\, \{v \neq \mathsf{unspec}\}$ is not valid, since it is not satisfied by any state σ for which $\sigma(v) = \mathbb{u}$.

Nor, *a fortiori*, does the bi-implication

$$\mathbb{K} \models \{p\}S\{q\} \iff \mathbb{K} \models \mathbf{sp}[p,S] \supset q$$

hold, at least not in the "$\Leftarrow$" direction, as we see by taking $q \equiv \mathbf{sp}[p,S]$.

However we have the following partial analogues to the results in 1.6.6. COROLLARIES.

(1) $\mathbb{K} \models \{p\}S\{q\} \Rightarrow (\mathbb{K} \models \{p\}S\{\mathbf{sp}[p,S]\}$ and $\mathbb{K} \models \mathbf{sp}[p,S] \supset q)$.

(2) (Intermediate assertion).

$$\mathbb{K} \models \{p\}S_1;S_2\{q\} \Rightarrow (\mathbb{K} \models \{p\}S_1\{\mathbf{sp}[p,S_1]\} \text{ and } \mathbb{K} \models \{\mathbf{sp}[p,S_1]\}S_2\{q\}).$$

PROOF. Exercise. The point is that if $A, \rho, \sigma \models p$ and $\mathcal{M}_A(S)(\sigma) = \sigma'$, then $A, \rho, \sigma' \models \mathbf{sp}[p,S_1]$, provided that $\sigma' \neq \varepsilon$! □

1.7 COMPLETENESS OF THE PROOF SYSTEM

1.7.1 Proof of completeness

We are now in a position to prove the converse of the soundness theorem (1.5.2).

THEOREM. *The system* $\mathbf{ProofSys}_{sa}(\mathbb{K})$ *is complete relative to* $\mathbb{K}$, *i.e. for any* $f \in \mathbf{Form}_{sa}(\Sigma)$,

$$\mathbb{K} \models f \Rightarrow \mathbb{K} \vdash f. \tag{1}$$

PROOF. If f is simply an assertion, then (1) is immediate, by the $\mathbb{K}$-oracle rule. So assume $f = \{p\}S\{q\}$. We will prove (1) by induction on $\mathbf{compl}(S)$.

Case 1. $S \equiv \mathbf{skip}$. Suppose $\mathbb{K} \models \{p\}\mathbf{skip}\{q\}$. Then $\mathbb{K} \models p \supset q$. So by the oracle rule, $\mathbb{K} \vdash p \supset q$. Hence by the '**skip**' rule, $\{p\}\mathbf{skip}\{q\}$.

Case 2. $S \equiv v := t$. Suppose $\{p\}v := t\{q\}$. Then by 1.6.6, Corollary 1, $\mathbb{K} \models p \supset (\mathbf{wp}[v := t, q])$, *i.e.*

$$\mathbb{K} \models p \supset (t \neq \mathbf{unspec} \wedge q\langle t/v\rangle).$$

Hence by the oracle rule,

$$\mathbb{K} \vdash p \supset (t \neq \mathbf{unspec} \wedge q\langle t/v\rangle),$$

Hence by the assignment rule (for simple variables), $\mathbb{K} \vdash \{p\}v := t\{q\}$.

Case 3. $S \equiv a[t_0] := t$. The argument is similar.

Case 4. $S \equiv S_1;S_2$. Suppose $\mathbb{K} \models \{p\}S_1;S_2\{q\}$. By 1.6.6, Corollaries 1

and 2,

$$\mathbb{K} \models \{p\} S_1 \{\mathbf{wp}[S_2, q]\} \quad \text{and} \quad \mathbb{K} \models \{\mathbf{wp}[S_2, q]\} S_2 \{q\}.$$

Hence by the induction hypothesis,

$$\mathbb{K} \vdash \{p\} S_1 \{\mathbf{wp}[S_2, q]\} \quad \text{and} \quad \mathbb{K} \vdash \{\mathbf{wp}[S_2, q]\} S_2 \{q\}.$$

Hence by the composition rule, $\mathbb{K} \vdash \{p\} S_1; S_2 \{q\}$.

Case 5. $S \equiv$ **if** b **then** S_1 **else** S_2 **fi**. Suppose

$$\mathbb{K} \models \{p\} S \{q\}. \tag{2}$$

We must show:

$$(a) \quad \mathbb{K} \models \{p \wedge (b = \mathbf{true})\} S_1 \{q\},$$

$$(b) \quad \mathbb{K} \models \{p \wedge (b = \mathbf{false})\} S_2 \{q\},$$

and

$$(c) \quad \mathbb{K} \models p \supset (b \neq \mathbf{unspec}),$$

since then $\mathbb{K} \vdash \{p\} S \{q\}$ follows by the induction hypothesis (applied to (a) and (b)), the oracle rule (applied to (c)) and the conditional rule.

For (a): take $(A, \rho, \sigma) \in \mathrm{INTERP}(\mathbb{K})$, and (dropping all reference to A and ρ from now on) suppose

$$\sigma \models p \wedge (b = \mathbf{true}). \tag{3}$$

Then

$$\sigma \models p \tag{4}$$

and

$$\mathcal{R}_A(b)(\sigma) = \mathrm{tt}. \tag{5}$$

Let

$$\sigma' = \mathcal{M}_A(S)(\sigma). \tag{6}$$

By (5),

$$\sigma' = \mathcal{M}_A(S_1)(\sigma) \tag{7}$$

and by (2), (4) and (6)

$$\sigma' \neq \varepsilon \quad \text{and} \quad \sigma' \models q. \tag{8}$$

Then (a) follows from (3), (6) and (8).

Similarly for (b). As for (c), take any $\sigma \neq \varepsilon$, and suppose $\sigma \models p$. Then $\mathcal{R}_A(b)(\sigma) = \mathrm{ff}$ would imply $\mathcal{M}_A(S)(\sigma) = \varepsilon$, contradicting (2). Hence $\mathcal{R}_A(b)(\sigma) \neq \mathrm{ff}$, and so (by 1.3.13) $\sigma \models b \neq \mathbf{unspec}$, thus proving (c), and Case 5, and the Theorem. □

Notice that the *consequence rule* was not needed in the completeness proof. Its redundancy here can be attributed to our special form of the assignment rule. For, with the form of this rule used in Apt [1981] or [dB], namely (considering assignments for simple variables) $\{p\langle t/v\rangle\} v := t \{p\}$, the consequence rule would indeed be needed.

1.7.2 Discussion: Use of the weakest precondition in the completeness proof

The expressibility of the weakest precondition was used in two places in the above proof: with the assignment rule (cases 2 and 3) and with the composition rule (case 4).

Now this use of the weakest precondition was not essential in the case of the *assignment* rules; it just simplified the proof. Actually the general notion of weakest precondition for an arbitrary statement was not used here, only the special case of an assignment statement. (In fact it could also have been used in the same way in the proof of the *soundness* theorem (1.5.2), to simplify the proof of the validity of the assignment rules. We avoided this, and introduced the notion of weakest precondition only after the proof of soundness, in order to emphasize that this notion is not necessary for that proof.

In the case of the *composition* rule, however, the expressibility of the weakest precondition (for an arbitrary statement) was used, in an essential way, in order to obtain an *intermediate assertion* ($\mathsf{wp}[S_2,q]$, in the above proof). In fact, we could also have obtained an intermediate assertion by use of the strongest postcondition ($\mathsf{sp}[p,S_1]$, in the notation of the proof).

Note that the numerical sort N was not needed in defining the assertions expressing the weakest precondition $\mathsf{wp}[S,p]$ or strongest postcondition $\mathsf{sp}[p,S]$, and was (therefore) not really needed for the theory of this Chapter. (Its use as the type of the subscripts of subscripted variables was just a matter of convenience; other types could have been chosen.) Thus *the results of this chapter hold for any class* $\mathbb{K}$, *with arbitrary signature, and not only for the standard classes.*

In Chapters 2 and 3, by contrast, the definition of the assertions expressing the weakest precondition and strongest postcondition will use the sort N, and the standard operations associated with it (see **1.1.1**, part 3(a)), in an essential way.

Chapter 2

'While' Programs

2.1 NOTATION FOR PARTIAL FUNCTIONS

Since partial functions will play an important role in the rest of this monograph, we collect some terminology and notation here.

(1) The notation '$f: A \dashrightarrow B$' means that f is a *partial function* from A to B, whereas '$f: A \to B$' means (usually) that f is a *total function*.

(2) For $f: A \dashrightarrow B$ and $x \in A$ we write $f(x)\downarrow$ ("$f(x)$ converges") if x is in the domain of f, and $f(x)\uparrow$ ("$f(x)$ diverges") otherwise. We also write $f(x)\downarrow y$ ("$f(x)$ converges to y") if $f(x)\downarrow$ and $f(x) = y$.

(3) For $f: A \dashrightarrow B$, $g: A \dashrightarrow B$ and $x \in A$, we write $f(x) \simeq g(x)$ if *either* $f(x)$ and $g(x)$ both converge, and to the same element of B, *or* they both diverge.

(4) We observe the usual convention for composing partial functions. Thus, for example, if $f: A \dashrightarrow B$ and $g: B \dashrightarrow C$, and we define $h: A \dashrightarrow C$ by $h(x) \simeq g(f(x))$, then $h(x)\uparrow$ if, and only if, *either* $f(x)\uparrow$ *or* $f(x)\downarrow y$ for some $y \in B$ such that $g(y)\uparrow$.

(5) By "function from A to B" we will generally mean: total function.

(6) We will also (in Chapter 3) consider "vector-valued partial functions"

$$\varphi: A \dashrightarrow B_1 \times \cdots \times B_k$$

with components $\varphi_i: A \dashrightarrow B_i \qquad (i = 1, \ldots, k)$,

where $\varphi(a) \simeq (\varphi_1(a), \ldots, \varphi_k(a))$

for $a \in A$. We will always assume that $\varphi(a)\downarrow$ if, and only if, $\varphi_i(a)\downarrow$ for *all* components i. In fact we will only work with vector-valued

functions φ which satisfy the (strong) assumption that

$$\varphi(a)\downarrow \text{ iff } \varphi_i(a)\downarrow \text{ for all } i,$$

and

$$\varphi(a)\uparrow \text{ iff } \varphi_i(a)\uparrow \text{ for all } i.$$

2.2 THE PROGRAMMING LANGUAGE

2.2.1 Syntax

The programming language $\boldsymbol{ProgLang}_{wa} = \boldsymbol{ProgLang}_{wa}(\Sigma)$ ('*wa*' for "while with arrays") is like the language $\boldsymbol{ProgLang}_{sa}(\Sigma)$ of Chapter 1, except that the class $\boldsymbol{Statemt}_{wa}$ of statements now also includes the 'while' statement, thus:

$$S ::= \textsf{skip} \mid v^i := t^i \mid a^i[t^{\mathbb{N}}] := t^i \mid S_1; S_2 \mid \textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \textsf{ fi} \mid \textsf{while } b \textsf{ do } S \textsf{ od}$$

For the rest, the definitions and notation of 1.2.1 and 1.2.2 concerning syntax carry over.

2.2.2 The programming language without arrays

We also consider the simpler language $\boldsymbol{ProgLang}_{w}(\Sigma)$ without arrays. All the concepts to be introduced in this chapter can be modified in an obvious way to apply to this language, and we refer to such concepts by using the subscript '*w*' instead of '*wa*'.

Although we only discuss the language with arrays in this chapter, we will use the simpler version in Chapters 3 and 4, when considering 'while' program computability.

2.2.3 Semantics

The semantics of *program terms* is exactly as in Chapter 1 (1.2.4—1.2.8). As for the semantics of *statements* (1.2.9—1.2.11), there is now an important difference, namely that execution of programs *need not terminate*. In fact we will define, for any $A \in \mathbb{K}$, a meaning function for statements

$$\mathcal{M}_A\colon \boldsymbol{Statemt}_{wa} \to (\mathrm{PR.STATE}(A) \dashrightarrow \mathrm{STATE}(A)),$$

so that $\mathcal{M}_A(S)$ is now a *partial function* from proper states to states (compare 1.2.11).

The idea is that the domain of $\mathcal{M}_A(S)$ consists of exactly those proper states σ for which execution of S, starting in σ, will eventually

terminate, either in a proper state or in the error state ε.

Again, for statements S and proper states σ, $\mathcal{M}_A(S)(\sigma)$ is defined by induction on ***compl***(S), as follows.

If S is **skip** or an assignment, then the definition is as before (1.2.11). Further

$$\mathcal{M}_A(S_1;S_2)(\sigma) \simeq \begin{cases} \mathcal{M}_A(S_2)(\sigma') & \text{if } \mathcal{M}_A(S_1)(\sigma)\downarrow\sigma' \neq \varepsilon, \\ \varepsilon & \text{if } \mathcal{M}_A(S_1)(\sigma)\downarrow\varepsilon, \\ \uparrow & \text{if } \mathcal{M}_A(S_1)(\sigma)\uparrow \end{cases}$$

$$\mathcal{M}_A(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi})(\sigma) \simeq \begin{cases} \mathcal{M}_A(S_1)(\sigma) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{t} \\ \mathcal{M}_A(S_2)(\sigma) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{f} \\ \varepsilon & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{u} \end{cases}$$

Finally let $S \equiv$ **while** b **do** S_0 **od**. Then there are three possibilities:

(*i*) *Normal termination*: $\mathcal{M}_A(S)(\sigma)\downarrow\sigma' \neq \varepsilon$ iff there exists $n \geq 0$ and $\sigma_0, \ldots, \sigma_n (\neq \varepsilon)$ such that $\sigma_0 = \sigma$ and $\sigma_n = \sigma'$ and for all $i < n$, $\mathcal{R}_A(b)(\sigma_i) = \mathbb{t}$ and $\mathcal{M}_A(S_0)(\sigma_i)\downarrow\sigma_{i+1}$, and $\mathcal{R}_A(b)(\sigma_n) = \mathbb{f}$.

(*ii*) *Exceptional termination* or *abortion*: $\mathcal{M}_A(S)(\sigma)\downarrow\varepsilon$ iff there exists $n \geq 0$ and $\sigma_0, \ldots, \sigma_n (\neq \varepsilon)$ such that $\sigma_0 = \sigma$ and for all $i < n$, $\mathcal{R}_A(b)(\sigma_i) = \mathbb{t}$ and $\mathcal{M}_A(S_0)(\sigma_i)\downarrow\sigma_{i+1}$, and *either* $\mathcal{R}_A(b)(\sigma_n) = \mathbb{u}$ *or* $\mathcal{R}_A(b)(\sigma_n) = \mathbb{t}$ and $\mathcal{M}_A(S_0)(\sigma_n)\downarrow\varepsilon$.

(*iii*) *Divergence*: $\mathcal{M}_A(S)(\sigma)\uparrow$ otherwise, *i.e.* iff *either* (*a*) there exist $n \geq 0$ and $\sigma_0, \ldots, \sigma_n (\neq \varepsilon)$ such that $\sigma_0 = \sigma$ and for all $i < n$, $\mathcal{R}_A(b)(\sigma_i) = \mathbb{t}$ and $\mathcal{M}_A(S_0)(\sigma_i) = \sigma_{i+1}$, and $\mathcal{M}_A(S_0)(\sigma_n)\uparrow$ (*local divergence*), *or* (*b*) there is an infinite sequence $\sigma_0, \sigma_1, \ldots (\neq \varepsilon)$ such that $\sigma_0 = \sigma$ and for all i, $\mathcal{R}_A(b)(\sigma_i) = \mathbb{t}$ and $\mathcal{M}_A(S_0)(\sigma_i)\downarrow\sigma_{i+1}$ (*global divergence*).

(Note that, above and elsewhere, we write '$\sigma_0, \ldots, \sigma_n (\neq \varepsilon)$' to mean: '$\sigma_0 (\neq \varepsilon), \ldots, \sigma_n (\neq \varepsilon)$'.)

2.2.4 Discussion

To repeat the difference between $\mathcal{M}_A(S)(\sigma) = \varepsilon$ and $\mathcal{M}_A(S)(\sigma)\uparrow$: The former means that execution of S, starting in state σ, terminates in the error state ε, or aborts, and this is always because some variable which was accessed in the course of this execution had the unspecified value $\mathbb{u}$ (as discussed in 1.2.11).

The latter means that execution of S, starting in σ, does not halt at all, but diverges; and it is easy to see that (in the case of the present language $\mathbf{ProgLang}_{wa}$) this is always ultimately due to global divergence of some 'while' statement contained in S (case $(iii)(b)$ in the definition of $\mathcal{M}_A$).

There is a sharp difference between the treatment of these two notions in the theory of partial correctness (see Section 2.4).

The definition in 2.2.3 gives an *operational semantics* for $\mathbf{Statemt}_{wa}$ (as in [dB], Definition 3.20). We could also give a *denotational semantics*, and prove their equivalence (as in [dB], Definition 3.22 and Theorem 3.25), but we refrain from this.

We *will* give both types of semantics for recursive programs in the next chapter, and prove their equivalence (3.1.10).

2.2.5 Computation sequences

To make our operational semantics clearer, we introduce the notion of computation sequence (as in [dB], p.148).

DEFINITION 1. A *computation sequence* over A is a finite or infinite (non-empty) sequence of states over A such that every state, except possibly the last (in the case of a finite sequence), is a proper state.

We let COMPSEQ(A) denote the set of computation sequences over A, with typical elements $\tau, \ldots$.

Thus a computation sequence has one of the forms:

$$\tau = (\sigma_0, \ldots, \sigma_{n-1}) \qquad (n>0),$$

or

$$\tau = (\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots).$$

In general $\sigma_i \neq \varepsilon$, except possibly in the first case, when $i = n-1$.

A computation sequence is intended to represent the "trace" of a computation, *i.e.* the sequence of all states, initial, intermediate and final, encountered during the execution of a statement.

DEFINITION 2. A finite computation sequence $(\sigma_0, \ldots, \sigma_{n-1})$ is *proper* if σ_{n-1} is a proper state (*i.e.* $\neq \varepsilon$), and *improper* otherwise.

Thus there are *three types* of computation sequence: *proper*, *improper* and *infinite* (the first two being finite). An improper computation sequence represents an aborted computation, and an infinite computation sequence represents a divergent computation.

We also define some (partial) operations on computation sequences:

DEFINITION 3. (*a*) The length $lh(\tau)$ of a sequence τ is defined by

$$lh(\tau) = \begin{cases} n & \text{if } \tau=(\sigma_0, \ldots, \sigma_{n-1}) \\ \infty & \text{otherwise.} \end{cases}$$

(*b*) The i-th component $\tau(i)$ of τ is defined by

$$\tau(i) = \begin{cases} \sigma_i & \text{if } \tau=(\sigma_0, \ldots, \sigma_i, \ldots) \text{ with } i < lh(\tau) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

(*c*) The last component $end(\tau)$ of τ is defined, for τ finite only, by:

$$end(\tau) = \tau(lh(\tau)-1).$$

Thus τ is *infinite* if $lh(\tau)=\infty$, *proper* if $lh(\tau)<\infty$ and $end(\tau)\neq\varepsilon$, and *improper* if $lh(\tau)<\infty$ and $end(\tau)=\varepsilon$.

DEFINITION 4. $\tau^\frown\tau'$, the *concatenation* of two computation sequences, is defined as follows. If τ is *infinite* or *improper*, then $\tau^\frown\tau'=\tau$. Otherwise, suppose $\tau=(\sigma_0, \ldots, \sigma_{n-1})$, with $\sigma_{n-1}\neq\varepsilon$. Then $\tau^\frown(\sigma_0', \sigma_1', \ldots)=(\sigma_0, \ldots, \sigma_{n-1}, \sigma_0', \sigma_1', \ldots)$.

DEFINITION 5. Let $lh(\tau)=l\leq\infty$.

(*a*) For $m\leq n<l$, $[\tau]_m^n$ is the segment $(\tau(m), \tau(m+1), \ldots, \tau(n))$, which is a computation sequence of length $n-m+1$.

(*b*) For $n<l$, $[\tau]_n$ is the tail-segment $[\tau]_n^l$ of τ.

2.2.6 The computation sequence generated by a statement from a state

We define a function

$$Comp_A: Statemt_{wa} \rightarrow (\text{PR.STATE}(A) \rightarrow \text{COMPSEQ}(A))$$

where $Comp_A(S)(\sigma)$ is the computation sequence generated by S, starting in state $\sigma(\neq\varepsilon)$ (compare [dB], Definition 5.16). We will see below that for all S and $\sigma\neq\varepsilon$,

$$end(Comp_A(S)(\sigma)) \simeq \mathcal{M}_A(S)(\sigma).$$

The definition is by induction on $compl(S)$:

(1) If S is skip or an assignment, then

$$Comp_A(S)(\sigma) = (\sigma, \mathcal{M}_A(S)(\sigma))$$

(*i.e.*, a sequence of length 2; see 1.2.11 for the definition of '$\mathcal{M}_A$').

(2) If $S \equiv S_1;S_2$: suppose $\mathbf{Comp}_A(S_1)(\sigma) = \tau$. Then

$$\mathbf{Comp}_A(S)(\sigma) = \begin{cases} \tau \hat{} \mathbf{Comp}_A(S_2)(\mathbf{end}(\tau)) & \text{if } \tau \text{ is proper} \\ \tau & \text{otherwise.} \end{cases}$$

(3) If $S \equiv$ **if** b **then** S_1 **else** S_2 **fi**, then

$$\mathbf{Comp}_A(S)(\sigma) = \begin{cases} (\sigma) \hat{} \mathbf{Comp}_A(S_1)(\sigma) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{t} \\ (\sigma) \hat{} \mathbf{Comp}_A(S_2)(\sigma) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{f} \\ (\sigma, \varepsilon) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{u}. \end{cases}$$

(4) If $S \equiv$ **while** b **do** S_0 **od**: there are four cases.

Case 1 (*Normal termination*). There exist $n \geq 0$ and proper computation sequences $\tau_1, \ldots, \tau_n$ such that (putting $\sigma_0 = \sigma$ and $\sigma_i = \mathbf{end}(\tau_i)$ for $1 \leq i \leq n$) for $i < n$, $\mathcal{R}_A(b)(\sigma_i) = \mathbb{t}$ and $\mathbf{Comp}_A(S_0)(\sigma_i) = \tau_{i+1}$, and $\mathcal{R}_A(b)(\sigma_n) = \mathbb{f}$. Then

$$\mathbf{Comp}_A(S)(\sigma) = (\sigma) \hat{} \tau_1 \hat{} \tau_2 \hat{} \cdots \hat{} \tau_n \hat{} (\sigma_n)$$

(Note that when $n = 0$ this reduces to (σ, σ)).

Case 2 (*Exceptional termination, type 1*). There exist $n \geq 0$ and proper computation sequences $\tau_1, \ldots, \tau_n$ such that (putting $\sigma_0 = \sigma$ and $\sigma_i = \mathbf{end}(\tau_i)$ for $1 \leq i \leq n$) for $i < n$, $\mathcal{R}_A(b)(\sigma_i) = \mathbb{t}$ and $\mathbf{Comp}_A(S_0)(\sigma_i) = \tau_{i+1}$, and $\mathcal{R}_A(b)(\sigma_n) = \mathbb{u}$. Then

$$\mathbf{Comp}_A(S)(\sigma) = (\sigma) \hat{} \tau_1 \hat{} \tau_2 \hat{} \cdots \hat{} \tau_n \hat{} (\varepsilon)$$

(Note that when $n = 0$ this reduces to (σ, ε)).

Case 3 (*Exceptional termination, type 2,* and *local divergence*). There exist $n > 0$ and $\tau_1, \ldots, \tau_n$ with $\tau_1, \ldots, \tau_{n-1}$ proper and τ_n *improper* or *infinite*, such that (putting $\sigma_0 = \sigma$ and $\sigma_i = \mathbf{end}(\tau_i)$ for $1 \leq i < n$) for $i < n$, $\mathcal{R}_A(b)(\sigma_i) = \mathbb{t}$ and $\mathbf{Comp}_A(S_0)(\sigma_i) = \tau_{i+1}$. Then

$$\mathbf{Comp}_A(S)(\sigma) = (\sigma) \hat{} \tau_1 \hat{} \tau_2 \hat{} \cdots \hat{} \tau_n .$$

Case 4 (*Global divergence*). There is an infinite sequence $\tau_1, \tau_2, \ldots$ (all proper) such that (putting $\sigma_0 = \sigma$ and $\sigma_i = \mathbf{end}(\tau_i)$ for all $i > 0$) for all $i \geq 0$, $\mathcal{R}_A(b)(\sigma_i) = \mathbb{t}$ and $\mathbf{Comp}_A(S_0)(\sigma_i) = \tau_{i+1}$. Then

$$\mathbf{Comp}_A(S)(\sigma) = (\sigma) \hat{} \tau_1 \hat{} \tau_2 \hat{} \cdots .$$

PROPOSITION. $\mathcal{M}_A(S)(\sigma) \simeq \mathbf{end}(\mathbf{Comp}_A(S)(\sigma))$.

PROOF. Induction on $\mathbf{compl}(S)$. □

The above proposition is clear, and not at all deep, since our definitions of $\mathcal{M}_A$ and $\mathbf{Comp}_A$ were both *operational.* In the next chapter, by contrast, a *denotational* or *fixed-point semantics* for programs is given, and the proof of equivalence between the two semantics is then less trivial (3.1.8).

2.2.7 Extensions of computation sequences

Let $\tau, \tau' \in \mathrm{COMPSEQ}(A)$ and $M \subseteq \mathbf{ProgVar}$. We have (compare 1.2.12):

DEFINITION. $\tau \subsetneq \tau'$ (rel M) iff *either*
(*i*) $\mathbf{lh}(\tau) = \mathbf{lh}(\tau')$ and for all $i < \mathbf{lh}(\tau)$, $\tau(i) \subsetneq \tau'(i)$ (rel M), *or*
(*ii*) $\mathbf{lh}(\tau) \leq \mathbf{lh}(\tau')$ and for all $i < \mathbf{lh}(\tau)$, $\tau(i) \subsetneq \tau'(i)$ (rel M), and $\mathbf{end}(\tau) = \varepsilon$.

2.2.8 Monotonicity for statements

(Compare 1.2.13.)

THEOREM. *Suppose* $M \supseteq \mathbf{ProgVar}(S)$, $\sigma, \sigma' \neq \varepsilon$, *and* $\sigma \subsetneq \sigma'$ (rel M). *Then*
(1) $\mathbf{Comp}_A(S)(\sigma) \subsetneq \mathbf{Comp}_A(S)(\sigma')$. *Hence:*
(2) *If* $\mathcal{M}_A(S)(\sigma)\downarrow \neq \varepsilon$ *then also* $\mathcal{M}_A(S)(\sigma')\downarrow \neq \varepsilon$ *and* $\mathcal{M}_A(S)(\sigma) \subsetneq \mathcal{M}_A(S)(\sigma')$ (rel M).
(3) *If* $\mathcal{M}_A(S)(\sigma)\uparrow$ *then also* $\mathcal{M}_A(S)(\sigma')\uparrow$.

PROOF of (1): Induction on $\mathbf{compl}(S)$. □

COROLLARY. *Suppose* $M \supseteq \mathbf{ProgVar}(S)$, $\sigma, \sigma' \neq \varepsilon$ *and* $\sigma \simeq \sigma'$ (rel M). *Then either*
(i) $\mathcal{M}_A(S)(\sigma)\downarrow \neq \varepsilon$, $\mathcal{M}_A(S)(\sigma')\downarrow \neq \varepsilon$ *and* $\mathcal{M}_A(S)(\sigma) \simeq \mathcal{M}_A(S)(\sigma')$ (rel M), *or*
(ii) $\mathcal{M}_A(S)(\sigma)\downarrow \varepsilon$ *and* $\mathcal{M}_A(S)(\sigma')\downarrow \varepsilon$, *or*
(iii) $\mathcal{M}_A(S)(\sigma)\uparrow$ *and* $\mathcal{M}_A(S)(\sigma')\uparrow$.

2.2.9 Variables in the "left hand side" of a statement; constancy of other variables

(Compare 1.2.14.) The definition of $\mathbf{lhs}(S)$ extends that in 1.2.14, with the clause

$$\mathbf{lhs}(\text{while } b \text{ do } S \text{ od}) = \mathbf{lhs}(S).$$

Again we have: $\mathbf{lhs}(S) \subseteq \mathbf{ProgVar}(S)$, and

THEOREM. *Suppose* $\sigma \neq \varepsilon$ *and* $\mathcal{M}_A(S)(\sigma)\downarrow \sigma' \neq \varepsilon$. *If* V *is not in* $\mathbf{lhs}(S)$ *then* $\sigma'(V) = \sigma(V)$.

PROOF. Induction on $\mathbf{compl}(S)$. □

*2.2.10 States which are specified on all relevant simple variables

The following proposition will be used in Chapter 4 (in 4.4.2; compare 1.2.15).

PROPOSITION 1. *Let S and $\sigma \neq \varepsilon$ be such that $\sigma(v) \neq \text{⫫}$ for all $v \in \textbf{\textit{SimVar}}(S)$, and $\textbf{\textit{ArrVar}}(S) = \emptyset$. Then either $\mathcal{M}_A(S)(\sigma)\uparrow$ or $\mathcal{M}_A(S)(\sigma)\downarrow \neq \varepsilon$, and, in the latter case, $\mathcal{M}_A(S)(\sigma)(v) \neq \text{⫫}$ for all $v \in \textbf{\textit{SimVar}}(S)$.*

PROOF. Induction on $\textbf{\textit{compl}}(S)$. Use 1.2.15, Proposition 1. □

The following is another version of this proposition, for the case that array variables may be present. The problem here is that it is then always possible for the program to abort, since (the evaluation of) an array variable may be unspecified at (the evaluation of) an index.

PROPOSITION 2. *Let S and $\sigma \neq \varepsilon$ be such that $\sigma(v) \neq \text{⫫}$ for all $v \in \textbf{\textit{SimVar}}(S)$. If $\mathcal{M}_A(S)(\sigma)\downarrow \neq \varepsilon$, then $\mathcal{M}_A(S)(\sigma)(v) \neq \text{⫫}$ for all $v \in \textbf{\textit{SimVar}}(S)$.*

PROOF. Induction on $\textbf{\textit{compl}}(S)$. □

2.2.11 Isomorphism between structures; semantics abstraction theorem

(Compare 1.2.16.)

THEOREM. *Given a Σ-isomorphism $\varphi: A \to B$ and any $S \in \textbf{\textit{Statemt}}(\Sigma)$ (and with the notation of 1.2.16), the following diagram commutes:*

$$\begin{array}{ccc} \text{PR.STATE}(A) & \xrightarrow{\mathcal{M}_A(S)} & \text{STATE}(A) \\ \Big\downarrow \hat{\varphi} & & \Big\downarrow \hat{\varphi}_\varepsilon \\ \text{PR.STATE}(B) & \xrightarrow{\mathcal{M}_B(S)} & \text{STATE}(B) \end{array}$$

PROOF. Induction on $\textbf{\textit{compl}}(S)$. □

So the semantics of our programming language $\textbf{\textit{ProgLang}}_{wa}(\Sigma)$ again satisfies the Program Semantics Abstraction Principle, or at least an obvious adaptation of the versions given earlier (0.3.2, 1.2.16) to *partial* meaning functions.

2.3 ASSERTIONS

2.3.1 Structures A^* of signature Σ^*

In order to be able to express the weakest precondition for **'while'** programs (2.6.11), it is necessary to strengthen the assertion language, by making it the first order language over an enriched signature Σ^*.

Consider a structure $A \in \mathbb{K}$ of signature Σ:

$$A = ((A_i)_{i \in Sort}, (F_j^A)_{1 \le j \le s}).$$

We have already defined (1.1.5) the structure A^u of signature Σ^u, on which our programming language semantics is based:

$$A^u = ((A_i^u)_{i \in Sort}, (F_j^{A,u})_{1 \le j \le s}, (\mathbb{u}_i)_{i \in Sort}).$$

Now we will describe the construction of an augmented structure A^* of signature Σ^*.

Define, for each sort i, the domain A_i^* to be the set of functions

$$\xi : \mathbb{N} \to A_i^u$$

such that the set

$$dom(\xi) =_{df} \{n \in \mathbb{N} \mid \xi(n) \neq \mathbb{u}\}$$

is finite.

We also have the new operations of application

$$\mathsf{Ap}_i^A : A_i^* \times \mathbb{N}^u \to A_i^u$$

where for $n \in \mathbb{N}$ $\quad \mathsf{Ap}_i^A(\xi, n) = \xi(n)$

and $\quad \mathsf{Ap}_i^A(\xi, \mathbb{u}_N) = \mathbb{u}_i.$

(We are continuing here with the convention of Chapter 1 of adding a superscript 'A' to a function symbol, to denote its interpretation relative to the structure A, or A^*. We are also continuing with the "boldface" convention 1.3.4.)

Then A^* is the structure obtained by adjoining the above to A^u, namely

$$A^* = ((A_i^u)_{i \in Sort}, (A_i^*)_{i \in Sort}, (F_j^{A,u})_{1 \le j \le s}, (\mathbb{u}_i)_{i \in Sort}, (\mathsf{Ap}_i^A)_{i \in Sort})$$

with signature Σ^*.

And $\mathbb{K}^*$ is the class of structures A^* for $A \in \mathbb{K}$.

2.3.2 Remarks

(1) The first-order language over Σ^* will be used as the assertion language for 'while' programs. Its value lies in the fact that it is possible, within this language, to formalize the semantics of the programming and assertion languages, and hence express the weakest precondition (2.6.11).

(2) The restriction that $\boldsymbol{dom}(\xi)$ be finite for $\xi \in A_i^*$ shows that we only need a form of *weak second order logic over* A.

(3) Since $A_{\mathbb{N}} = \mathbb{N}$ and $A_{\mathbb{B}} = \mathbb{B}$, it is not really necessary to have separate domains $A_{\mathbb{N}}^*$ and $A_{\mathbb{B}}^*$, since these could easily be coded in $\mathbb{N}$ (as in Zucker [1980]). However we keep these domains for the sake of uniformity of treatment with the algebraic domains.

2.3.3 The assertion language: syntax

The assertion language $\boldsymbol{AssLang}_{wa} = \boldsymbol{AssLang}_{wa}(\Sigma)$ is $\boldsymbol{Lang}_1(\Sigma^*)$, the first-order language over Σ^*. The details are as follows.

For each sort i, we have *two types*: i and i^*, representing the domains A_i^{μ} and A_i^* respectively. (So the type i corresponds exactly to the sort i of Chapter 1.)

The language then extends the language $\boldsymbol{AssLang}_{sa}$ of Chapter 1 (1.3.1) correspondingly, with the following syntactic classes:

(1) In addition to the class $\boldsymbol{AssVar}_i$ of assertion variables of type i, there is also a class $\boldsymbol{AssVar}_i^*$ of assertion variables of type i^*, denoted $\xi^i, \eta^i, \zeta^i, \ldots$. We then define

$$\boldsymbol{AssVar} =_{df} \bigcup_{i \in \boldsymbol{Sort}} \boldsymbol{AssVar}_i \cup \bigcup_{i \in \boldsymbol{Sort}} \boldsymbol{AssVar}_i^*.$$

(2) $\boldsymbol{AssTerm}_i$, the class of assertion terms of type i, again denoted $s^i, \ldots$, is defined as in Chapter 1 (1.3.1,(2)), but with one more formation rule, using the application operation:

$$s^i ::= \cdots \mid \mathsf{Ap}_i(\xi^i, s^{\mathbb{N}}).$$

We will usually write '$\xi(s)$' for '$\mathsf{Ap}(\xi, s)$'.

(3) $\boldsymbol{Assn}^* = \boldsymbol{Assn}^*(\Sigma)$, the class of assertions, again denoted $p, q, r, \ldots$, is defined like $\boldsymbol{Assn}$ (1.3.1,(3)), but with one more formation rule, for quantification over A_i^*:

$$p ::= \cdots \mid \exists \xi^i [p].$$

We do not need equalities between assertion variables of type i^*, since we can take the formula '$\xi_1^i = \xi_2^i$' to be an abbreviation for '$\forall z^{\mathbb{N}}[\xi_1(z^{\mathbb{N}}) = \xi_2(z^{\mathbb{N}})]$'.

2.3.4 Notation

Regarding the assertion language:

(1) We will use $z^{\mathsf{N}}, \ldots$ or $z, \ldots$ exclusively for variables of type N. Correspondingly, we will use $\zeta^{\mathsf{N}}, \ldots$ or $\zeta, \ldots$ for assertion variables of type N^* only.

(2) We will use $i, j, k, l, m, n, \ldots$ to range over natural numbers, thus:

$$\forall n[\cdots n \cdots] \quad \text{means} \quad \forall z[z \neq \mathsf{unspec}_{\mathsf{N}} \supset \cdots z \cdots]$$

and

$$\exists n[\cdots n \cdots] \quad \text{means} \quad \exists z[z \neq \mathsf{unspec}_{\mathsf{N}} \wedge \cdots z \cdots].$$

(Remember, the assertion variables of type i range over A_i^{μ}, not A_i.)

2.3.5 The assertion language: semantics

We are now interested in valuations over A^*, where (*cf.* 1.3.5) a *valuation over* A^* is defined to be a function of the form

$$\rho = \bigcup_{i \in \mathit{Sort}} (\rho_i \cup \rho_i^*)$$

where for each sort i

$$\rho_i : \mathit{AssVar}_i \to A_i^{\mu}$$

and

$$\rho_i^* : \mathit{AssVar}_i^* \to A_i^*.$$

The set of such valuations is called $\mathrm{VAL}(A^*)$, with elements denoted by $\rho, \ldots$.

The semantics of the *assertion terms* and *assertions* is then given by modifying the definitions of the functions $\mathcal{S}_A$ (1.3.7) and $\mathcal{J}_A$ (1.3.10) respectively in an obvious way.

An *interpretation in* $\mathbb{K}^*$ (*cf.* 1.3.11) is now defined as a triple $I = (A^*, \rho, \sigma)$ where $A \in \mathbb{K}$ (and hence $A^* \in \mathbb{K}^*$), $\rho \in \mathrm{VAL}(A^*)$ and (as before) $\sigma \in \mathrm{PR.STATE}(A)$. The class of all such interpretations is called $\mathrm{INTERP}^*(\mathbb{K})$.

The notions of *satisfaction*, $I \models p$, and $\mathbb{K}$-*validity*, $\mathbb{K} \models p$, are defined as in 1.3.11, where now $\mathbb{K} \models p$ iff $I \models p$ for every $I \in \mathrm{INTERP}^*(\mathbb{K})$.

All the results of Section 1.3 then carry over in an obvious way.

2.4 CORRECTNESS FORMULAE

2.4.1 Syntax

The class $\boldsymbol{Form}_{wa} = \boldsymbol{Form}_{wa}(\Sigma)$ of *correctness formulae* (relative to Σ), denoted $f, \ldots$, is defined as in Chapter 1 (1.4.1):

$$f ::= \{p\}S\{q\} \mid p\,.$$

2.4.2 Semantics

Their semantics, however, differs significantly from that in Chapter 1, since $\mathcal{M}_A(S)$ is now a *partial function* on states. (Compare 1.4.2.)

DEFINITIONS. (1) (*Satisfaction.*) We define, for an interpretation $I = (A^*, \rho, \sigma)$ and correctness formula f, the notion $I \models f$ as follows.

Case 1. $f \equiv \{p\}S\{q\}$. Then $A^*, \rho, \sigma \models f$ iff for all σ':

$$(A^*, \rho, \sigma \models p \;\; \textit{and} \;\; \mathcal{M}_A(S)(\sigma)\downarrow\sigma') \;\Rightarrow\; (\sigma' \neq \varepsilon \;\text{ and }\; A^*, \rho, \sigma' \models q\,).$$

Case 2. $f \equiv p$. Then $I \models f$ as in 2.3.5.

(2) ($\mathbb{K}$-*validity.*) $\mathbb{K} \models f$ iff $I \models f$ for all $I \in \mathrm{INTERP}^*(\mathbb{K})$.

2.4.3 Discussion

In the definition of '$I \models \{p\}S\{q\}$', notice that the condition that $\mathcal{M}_A(S)(\sigma)\downarrow$ is included in the *assumption*, but the condition that $\mathcal{M}_A(S)(\sigma) \neq \varepsilon$ is in the *conclusion*. Thus, for example, for $I = (A^*, \rho, \sigma)$,

$$I \models \{\mathsf{true}\}S\{q\} \;\text{ if }\; \mathcal{M}_A(S)(\sigma)\uparrow,$$

but

$$I \not\models \{\mathsf{true}\}S\{q\} \;\text{ if }\; \mathcal{M}_A(S)(\sigma)\downarrow\varepsilon.$$

So this notion of *partial correctness* distinguishes sharply between a program execution which never terminates (diverges), and one which terminates in an error state (aborts). (The semantics of *total correctness* is considered in Section 2.8.)

2.5 A PROOF SYSTEM; SOUNDNESS

2.5.1 The proof system

We define a proof system $\boldsymbol{ProofSys}_{wa} = \boldsymbol{ProofSys}_{wa}(\mathbb{K})$ for deriving $\mathbb{K}$-valid correctness formulae. This is just like the system $\boldsymbol{ProofSys}_{sa}$ of Chapter 1 (1.5.1), except for the addition of a rule for **'while'** statements:

(A.5) *The* **'while'** *rule*:

$$\frac{p \supset r, \quad \{r \wedge (b = \mathsf{true})\} S \{r\}, \quad r \wedge (b = \mathsf{false}) \supset q, \quad r \supset (b \neq \mathsf{unspec})}{\{p\}\,\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}\{q\}}$$

The assertion r is called an *invariant* for the **'while'** statement, or a *loop invariant* (with respect to p and q).

Now we use the notation '$\mathbb{K} \vdash f$' for derivability of f in $\boldsymbol{ProofSys}_{wa}(\mathbb{K})$.

We prove soundness of this system below, and completeness later (Section 2.7).

2.5.2 Soundness

THEOREM. *The system* $\boldsymbol{ProofSys}(\mathbb{K})$ *is sound relative to* $\mathbb{K}$, *i.e. for any* $f \in \boldsymbol{Form}_{wa}(\Sigma)$,

$$\mathbb{K} \vdash f \ \Rightarrow\ \mathbb{K} \models f.$$

PROOF. This amounts to showing (as in 1.5.2) that each inference rule is $\mathbb{K}$-valid, *i.e.* $\mathbb{K}$-validity of the premisses implies $\mathbb{K}$-validity of the conclusion. However it is now *not* sufficient just to check the new **'while'** rule, as there are complications even for other rules (since $\mathcal{M}_A(S)$ is no longer a total function).

The arguments for the **'skip'** and assignment rules *do* carry over exactly from 1.5.2, since for $S \equiv \mathsf{skip}$ or an assignment, $\mathcal{M}_A(S)(\sigma)$ *is* a total function. Consider now the composition rule:

$$\frac{\{p\}S_1\{r\}, \quad \{r\}S_2\{q\}}{\{p\}S_1;S_2\{q\}}\ .$$

Suppose

$$\mathbb{K} \models \{p\}S_1\{r\} \tag{1}$$

and

$$\mathbb{K} \models \{r\}S_1\{q\}. \tag{2}$$

We must show: $\models \{p\}S_1;S_2\{q\}$. So take an interpretation $I = (A^*, \rho, \sigma)$ in $\mathbb{K}^*$ such that (suppressing the 'A^*' and 'ρ')

$$\sigma \models p, \tag{3}$$

and suppose

$$\mathcal{M}_A(S_1;S_2)(\sigma) \downarrow \sigma'. \tag{4}$$

We must show:

$$\sigma' \neq \varepsilon \ \text{ and } \ \sigma' \models q.$$

Now $\mathcal{M}_A(S_1)(\sigma)\uparrow$ would imply $\mathcal{M}_A(S_1;S_2)(\sigma)\uparrow$, contradicting (4). So suppose

$$\mathcal{M}_A(S_1)(\sigma)\downarrow\sigma''. \tag{5}$$

By (1) and (3), $\sigma'' \neq \varepsilon$ (6)

and $\sigma'' \models r.$ (7)

By (4), (5), (6) and the definition of $\mathcal{M}_A$,

$$\mathcal{M}_A(S_2)(\sigma'')\downarrow\sigma'. \tag{8}$$

Finally, by (2), (7) and (8), $\sigma' \neq \varepsilon$ and $\sigma' \models q$.

The conditional and consequence rules are left as exercises. Now consider the 'while' rule. Suppose

$$\mathbb{K} \models p \supset r, \tag{9}$$

$$\mathbb{K} \models \{r \wedge (b = \textsf{true})\} S \{r\}, \tag{10}$$

$$\mathbb{K} \models r \wedge (b = \textsf{false}) \supset q \tag{11}$$

and $$\mathbb{K} \models r \supset (b \neq \textsf{unspec}). \tag{12}$$

We must show $\models \{p\} S_1 \{q\}$, where

$$S_1 \equiv \textsf{while } b \textsf{ do } S \textsf{ od}.$$

So take an interpretation $I = (A^*, \rho, \sigma)$ in $\mathbb{K}^*$ such that

$$\sigma \models p, \tag{13}$$

and suppose $$\mathcal{M}_A(S_1)(\sigma)\downarrow\sigma'. \tag{14}$$

We must show: $\sigma' \neq \varepsilon$ and $\sigma' \models q$.

Now consider the definition of $\mathcal{M}_A(S_1)$ (in 2.2.3). By (14), either case (*i*) or case (*ii*) (*loc. cit.*) holds. In other words: there exist $n \geq 0$ and $\sigma_0, \ldots, \sigma_n (\neq \varepsilon)$ such that $\sigma_0 = \sigma$, and for all $i < n$

$$\mathcal{R}_A(b)(\sigma_i) = \mathbb{t} \tag{15}$$

and $$\mathcal{M}_A(S)(\sigma_i)\downarrow\sigma_{i+1}, \tag{16}$$

and *either*
(*a*) $\mathcal{R}_A(b)(\sigma_n) = \mathbb{t}$ and $\mathcal{M}_A(S)(\sigma_n)\downarrow\sigma' \neq \varepsilon$ and $\mathcal{R}_A(b)(\sigma') = \mathbb{f}$, *or*
(*b*) $\mathcal{R}_A(b)(\sigma_n) = \mathbb{t}$ and $\mathcal{M}_A(S)(\sigma_n)\downarrow\varepsilon$ and $\sigma' = \varepsilon$, *or*
(*c*) $\mathcal{R}_A(b)(\sigma_n) = \mathbb{u}$ and $\sigma' = \varepsilon$.

Now by (9) and (13), $\sigma \models r.$

Then by repeated application of (10), (15) and (16), for all $i < n$,

$$\sigma_i \models r \wedge (b = \mathbf{true}).$$

By one more application of (10), (15) and (16),

$$\sigma_n \models r. \tag{17}$$

By (17), (12) and Proposition 1.3.13, $\mathcal{R}_A(b)(\sigma_n) \neq \mathbb{u}$. This excludes case (*c*) above. By the remaining cases (*a*) and (*b*),

$$\mathcal{R}_A(b)(\sigma_n) = \mathbb{t}. \tag{18}$$

By (17), (18) and (10), case (*b*) is excluded, and further, by (*a*), $\sigma' \neq \varepsilon$ and

$$\sigma' \models r \tag{19}$$

and

$$\mathcal{R}_A(b)(\sigma') = \mathbb{f}. \tag{20}$$

So by (11), (19) and (20), $\sigma' \models q$. □

2.6 PARTIAL STATE TRANSFORMERS; THE WEAKEST PRECONDITION AND STRONGEST POSTCONDITION

2.6.1 Partial state transformers

The notion of a *state predicate* on a structure $A \in \mathbb{K}$ is exactly as defined in Chapter 1 (1.6.1). However we are now interested in *partial* state transformers (compare 1.6.2).

DEFINITIONS. (1) A *partial state transformer* on A is a partial function

$$\Phi\colon \text{PR.STATE}(A) \dashrightarrow \text{STATE}(A)$$

which is *monotone*, in the following sense. Suppose $\sigma, \sigma' \in \text{PR.STATE}(A)$ and $\sigma \subsetneq \sigma'$ (see 1.2.12, Definition 1). Then
(*i*) if $\Phi(\sigma)\downarrow \neq \varepsilon$ then $\Phi(\sigma')\downarrow \neq \varepsilon$ and $\Phi(\sigma) \subsetneq \Phi(\sigma')$, and
(*ii*) if $\Phi(\sigma)\uparrow$ then $\Phi(\sigma')\uparrow$.

(Note that it is permitted that $\Phi(\sigma)\downarrow \varepsilon$ and $\Phi(\sigma')\uparrow$.)

(2) A partial state transformer Φ is *finitely based* if there exists a finite set $M \subset \mathbf{ProgVar}$ such that
(*a*) for all $\sigma, \sigma' \in \text{PR.STATE}(A)$, if $\sigma \simeq \sigma'$ (rel M) then *either*
  (*i*) $\Phi(\sigma)\downarrow \neq \varepsilon$ and $\Phi(\sigma')\downarrow \neq \varepsilon$ and $\Phi(\sigma) \simeq \Phi(\sigma')$ (rel M), *or*
  (*ii*) $\Phi(\sigma)\downarrow \varepsilon$ and $\Phi(\sigma')\downarrow \varepsilon$, *or*
  (*iii*) $\Phi(\sigma)\uparrow$ and $\Phi(\sigma')\uparrow$; and
(*b*) for all V *not* in M and all $\sigma \in \text{PR.STATE}(A)$,

$$\Phi(\sigma)\downarrow \neq \varepsilon \;\Rightarrow\; \Phi(\sigma)(V) = \sigma(V).$$

The remark in 1.6.2, on equivalent formulations of the monotonicity and finite determinateness conditions, also applies here (when modified in the obvious way to allow for divergence).

PROPOSITION. *For any statement S, $\mathcal{M}_A(S)$ is a finitely based partial state transformer.*

PROOF. By 2.2.8 and 2.2.9. □

2.6.2 Weakest precondition and strongest postcondition

(Compare 1.6.3.)

DEFINITIONS. (1) (*Weakest precondition.*) Given a partial state transformer Φ and a predicate π, both on A, the *weakest precondition* of Φ and π, written $\mathrm{WP}_A(\Phi, \pi)$, is the predicate on A which holds at $\sigma \in \mathrm{PR.STATE}(A)$ iff

$$\Phi(\sigma)\downarrow \;\Rightarrow\; (\Phi(\sigma) \neq \varepsilon \text{ and } \pi(\Phi(\sigma)) = \mathbb{t}).$$

In other words, $\mathrm{WP}_A(\Phi, \pi)(\sigma) = \mathbb{t}$ iff

either $\Phi(\sigma)\downarrow \neq \varepsilon$ and $\pi(\Phi(\sigma)) = \mathbb{t}$,

or $\Phi(\sigma)\uparrow$.

(2) (*Strongest postcondition.*) Given a state transformer Φ and predicate π, both on A, the *strongest postcondition* of Φ and π, written $\mathrm{SP}_A(\pi, \Phi)$, is the predicate on A which holds at $\sigma \in \mathrm{PR.STATE}(A)$ iff

$$\text{there exists } \sigma' \neq \varepsilon \text{ such that } \pi(\sigma') = \mathbb{t} \text{ and } \Phi(\sigma') = \sigma.$$

Again, we have:

PROPOSITION. *If Φ and π are finitely based, then so are $\mathrm{WP}_A(\Phi, \pi)$ and $\mathrm{SP}_A(\pi, \Phi)$.*

Note that $\mathrm{WP}_A(\Phi, \pi)$ is the weakest precondition for *partial correctness* (called "weakest liberal precondition" in Dijkstra [1976]), which is what we emphasize in this monograph. The weakest precondition for *total correctness* (called simply "weakest precondition" in Dijkstra [1976]) is discussed briefly in the last sections of this, and the next, chapter.

2.6.3 Expressibility of the weakest precondition and strongest postcondition: Introduction

The notion of $\mathbb{K}$-*expressibility* of the weakest precondition and strongest postcondition is defined as in 1.6.4.

The rest of this section (2.6) is devoted to a proof of this expressibility (Theorems 4 and 5 in 2.6.11). Some preparation is needed for this, as we now briefly explain.

The main step in this proof is the proof of the expressibility of the *computation predicate* in the assertion language $\boldsymbol{AssLang}_{wa}$ (Definition 2 and Theorem 2 in 2.6.11).

In building up to this, we will see how to formalize, within this language, the notions of state (2.6.5/6) and computation sequence (2.6.9/10). It is here that we will see why the assertion language was defined as it was, *i.e.* as a weak second order language!

The central issue, briefly, is the representability of finite sequences (*e.g.* computation sequences) over the domains. Now for the domain of natural numbers (or integers) this can be done in a well-known way, within a first-order language over the structure, by a primitive recursive coding of finite sequences of numbers as single numbers. However, over abstract structures this is, in general, impossible, and we need a separate type of finite sequences (or finite functions) over each domain.

2.6.4 Important convention

Assume, from now on, that all program terms, statements, assertion terms and assertions, with which we deal, contain simple and array variables only among $\vec{v}, \vec{a}$, where

$\vec{v}$ is the tuple $(v_1, \ldots, v_{M_1})$ of sorts $(k_1, \ldots, k_{M_1})$ respectively,
and $\vec{a}$ is the tuple $(a_1, \ldots, a_{M_2})$ of sorts $(l_1, \ldots, l_{M_2})$ respectively.

2.6.5 Representation of states by vectors in A^*

With the above convention, we can represent a state σ over A by a finite sequence or *vector* of elements of A^*:

$$X = (x_0^{\mathbf{B}}, x_1, \ldots, x_{M_1}, \xi_1, \ldots, \xi_{M_2}) \qquad (*)$$

where M_1 and M_2 are as in 2.6.4,
$x_0^{\mathbf{B}} \in \mathbb{B}^{\mathbf{u}}$,
$x_i \in A_{k_i}^{\mathbf{u}}$ for $i = 1, \ldots, M_1$,
and $\xi_j \in A_{l_j}^*$ for $j = 1, \ldots, M_2$.

The idea is that x_0^{B} is a "flag", indicating whether σ is a proper state or not. In the former case, $x_i = \sigma(v_i)$ (=$\mathbb{u}$ possibly) and $\xi_j(n) = \sigma(a_j, n)$ (=$\mathbb{u}$ possibly) for $1 \le i \le M_1$, $1 \le j \le M_2$ and $n \in \mathbb{N}$. More precisely:

DEFINITIONS. (1) Let $\vec{k}$ be the sequence of types $k_1, \ldots, k_{M_1}$, and $\vec{l}^*$ the sequence of types $l_1^*, \ldots, l_{M_2}^*$. Then $A^*[\mathsf{B}, \vec{k}, \vec{l}^*]$ is the product domain

$$\mathbb{B}^{\mathbb{u}} \times A_{k_1}^{\mathbb{u}} \times \cdots \times A_{k_{M_1}}^{\mathbb{u}} \times A_{l_1}^* \times \cdots \times A_{l_{M_2}}^*$$

with typical elements $\boldsymbol{X}, \ldots$, *i.e. vectors*, as in (*) above.

(2) The *error vector* error^A in A^* is defined as

$$(\mathbb{u}_{\mathsf{B}}, \mathbb{u}_{k_1}, \ldots, \mathbb{u}_{k_{M_1}}, \lambda n \cdot \mathbb{u}_{l_1}, \ldots, \lambda n \cdot \mathbb{u}_{l_{M_2}})$$

in $A^*[\mathsf{B}, \vec{k}, \vec{l}^*]$, which represents the error state ε.

(3) For a vector $\boldsymbol{X}$ as in (*), and a state σ over A, $\boldsymbol{X}$ *represents* σ iff *either* (*a*) $\sigma = \varepsilon$ and $\boldsymbol{X} = \mathsf{error}^A$, *or* (*b*) $\sigma \neq \varepsilon$, $\boldsymbol{x}_0^{\mathsf{B}} = \mathbb{t}$ and $\boldsymbol{x}_i = \sigma(v_i)$ and $\boldsymbol{\xi}_j(n) = \sigma(a_j, n)$ for $1 \le i \le M_1$, $1 \le j \le M_2$ and $n \in \mathbb{N}$.

(4) $\ulcorner \sigma \urcorner$ is the unique vector which represents σ.

Note that by 1.3.9 (Corollary), 1.3.12, 2.2.8 (Corollary) and 2.2.9, we could assume that all our proper states σ are completely *unspecified* on all program variables other than $\vec{v}$ and $\vec{a}$. In that case, the vector $\ulcorner \sigma \urcorner$ gives an *exact* representation of σ, and we could even identify σ and $\ulcorner \sigma \urcorner$.

2.6.6 Representation of states formally in the assertion language

Consider now a *tuple* of assertion variables

$$X \equiv (x_0^{\mathsf{B}}, x_1, \ldots, x_{M_1}, \xi_1, \ldots, \xi_{M_2}) \qquad (**)$$

where x_i has type k_i for $i = 1, \ldots, M_1$,
and ξ_j has type l_j^* for $j = 1, \ldots, M_2$.

We will use $X, \ldots$ for tuples of assertion variables as in (**).

Now take $\rho \in \mathrm{VAL}(A^*)$ and a tuple X as in (**).

DEFINITIONS. (1) $\rho(X)$ denotes the vector

$$(\rho(x_0^{\mathsf{B}}), \rho(x_1), \ldots, \rho(x_{M_1}), \rho(\xi_1), \ldots, \rho(\xi_{M_2})).$$

(2) X *represents* a state σ relative to ρ iff $\rho(X)$ represents σ, *i.e.*, $\rho(X) = \ulcorner \sigma \urcorner$.

(3) (*Variant* of a valuation.) $\rho\{\boldsymbol{X}/X\}$ is the valuation ρ' which agrees with ρ off X, and such that $\rho'(X) = \boldsymbol{X}$.

2.6.7 Representation of the semantics of assertion terms and assertions within the assertion language

Let $X \equiv (x_0^{\mathsf{B}}, x_1, \ldots, x_{M_1}, \xi_1, \ldots, \xi_{M_2})$ be a tuple of assertion variables as in (**) above.

We will define, for each assertion term s and assertion p (where $\boldsymbol{Var}(s)$ and $\boldsymbol{Var}(p)$ are disjoint from X) another term $s[\![X]\!]$, and assertion $p[\![X]\!]$, which "represent" the semantics $\mathcal{S}_A$ of s, and $\mathcal{T}_A$ of p, respectively, relative to the state "represented by" X, in the sense of the theorem stated below.

The idea is that the simple and array program variables v_i and a_j in these expressions are replaced, respectively, by the assertion variables x_i and ξ_j in X.

More precisely:

DEFINITION 1. Let s be an assertion term such that

$$\boldsymbol{AssVar}(s) \cap X = \emptyset. \tag{1}$$

We define an assertion term $s[\![X]\!]$, of the same type as s, by induction on $\boldsymbol{compl}(s)$. (The first two cases are the interesting ones; the rest are trivial.)

$$v_i[\![X]\!] \equiv x_i \qquad (1 \le i \le M_1)$$

$$a_j[s][\![X]\!] \equiv \xi_j(s[\![X]\!]) \qquad (1 \le j \le M_2)$$

$$x[\![X]\!] \equiv x \qquad \text{(Remember, by (1), that } x \text{ and } \xi \text{ are not in } X.\text{)}$$

$$\xi(s)[\![X]\!] \equiv \xi(s[\![X]\!])$$

$$F(s_1, \ldots, s_m)[\![X]\!] \equiv F(s_1[\![X]\!], \ldots, s_m[\![X]\!])$$

$$\textsf{if } s^{\mathsf{B}} \textsf{ then } s_1 \textsf{ else } s_2 \textsf{ fi}[\![X]\!] \equiv \textsf{if } s^{\mathsf{B}}[\![X]\!] \textsf{ then } s_1[\![X]\!] \textsf{ else } s_2[\![X]\!] \textsf{ fi}$$

$$\textsf{unspec}[\![X]\!] \equiv \textsf{unspec}.$$

REMARK. It is clear that

$$\boldsymbol{ProgVar}(s[\![X]\!]) = \emptyset$$

and $$\boldsymbol{AssVar}(s) \subseteq \boldsymbol{AssVar}(s[\![X]\!]) \subseteq \boldsymbol{AssVar}(s) \cup X.$$

DEFINITION 2. Let p be an assertion such that

$$\boldsymbol{AssVar}(p) \cap X = \emptyset. \tag{2}$$

(where, as in Chapter 1, $\boldsymbol{AssVar}(p)$ refers to the *free* assertion variables in p). We will define an assertion $p[\![X]\!]$.

First let us assume that p satisfies the stronger condition:

no *free* or *bound* assertion variables in p are in X. (2′)

The definition of $p[\![X]\!]$ is by induction on $\mathbf{compl}(p)$. (Now all the cases are trivial. Note that any sub-assertion of an assertion satisfying (2′) also satisfies (2′).)

$$(s_1 = s_2)[\![X]\!] \equiv (s_1[\![X]\!] = s_2[\![X]\!])$$

$$(\neg p)[\![X]\!] \equiv \neg(p[\![X]\!])$$

$$(p_1 \wedge p_2)[\![X]\!] \equiv p_1[\![X]\!] \wedge p_2[\![X]\!]$$

$$\exists x\,[p][\![X]\!] \equiv \exists x\,[p[\![X]\!]] \qquad \text{(Remember, by (2′), that}$$

$$\exists \xi\,[p][\![X]\!] \equiv \exists \xi\,[p[\![X]\!]] \qquad x \text{ and } \xi \text{ are not in } X.\text{)}$$

Now let p be any assertion which satisfies (2). Then we define

$$p[\![X]\!] \equiv_{df} p'[\![X]\!]$$

where p' is an assertion *congruent* to p (see 1.3.17) which satisfies (2′).

REMARKS. (1) $p[\![X]\!]$ is well defined up to congruence, *i.e.* if p' and p'' are both congruent to p and satisfy (2′), then $p'[\![X]\!] \cong p''[\![X]\!]$. (We omit the proof.)

(2) It is clear that

$$\mathbf{ProgVar}(p[\![X]\!]) = \emptyset$$

and $$\mathbf{AssVar}(p) \subseteq \mathbf{AssVar}(p[\![X]\!]) \subseteq \mathbf{AssVar}(p) \cup X.$$

Hence we may use the notation of the Remarks in 1.3.9 and 1.3.12 in the following Theorem.

THEOREM. *Suppose X represents $\sigma(\neq\varepsilon)$ relative to ρ. If $\mathbf{Var}(s)$ and $\mathbf{Var}(p)$ are disjoint from X, then*

(*a*) $\mathcal{S}_A(s[\![X]\!])(\rho,\cdot) = \mathcal{S}_A(s)(\rho,\sigma)$,

(*b*) $\mathcal{T}_A(p[\![X]\!])(\rho,\cdot) = \mathcal{T}_A(p)(\rho,\sigma)$.

To restate (b): For any interpretation (A^*,ρ,σ) and tuple X such that X represents σ relative to ρ, and any p with $\mathbf{Var}(p) \cap X = \emptyset$,

$$A^*, \rho \vDash p[\![X]\!] \iff A^*, \rho, \sigma \vDash p.$$

PROOF. Induction on $\mathbf{compl}(s)$ and $\mathbf{compl}(p)$. □

2.6.8 Functions of two variables in A^*

In order to represent finite computation sequences within the language of A^*, we must also consider, for each sort i, *binary functions, i.e.* functions of *two* number variables

$$\eta\colon \mathbb{N}^2 \to A_i^{\mathsf{u}},$$

again with the restriction that $\boldsymbol{dom}(\eta) =_{df} \{(m,n) \mid \eta(m,n) \neq \mathbb{u}\}$ is finite.

This involves extending the language to include (for each sort i) assertion variables η^i for binary functions, and the corresponding application operator $\mathsf{Ap}_i^{(2)}$, with the new formation rule

$$s^i ::= \cdots \mid \mathsf{Ap}_i^{(2)}(\eta^i, s_1^{\mathsf{N}}, s_2^{\mathsf{N}}).$$

We let $\eta^i, \ldots$ or $\eta, \ldots$ denote assertion variables for binary functions. Also we write '$\eta(s_1, s_2)$' instead of '$\mathsf{Ap}^{(2)}(\eta, s_1, s_2)$'.

Corresponding to this extension of the assertion language $\boldsymbol{AssLang}_{wa}$, there is an extension of the structure A^* to include, for each sort i, a domain $A_i^{*(2)}$ of binary functions η^i.

However this provides an *inessential* extension of $\boldsymbol{AssLang}_{wa}$, or (to put the matter another way) the domain $A_i^{*(2)}$ can be modelled within A_i^*, by coding a binary function η in the well-known way as a unary function $\xi\colon \mathbb{N} \to A_i^{\mathsf{u}}$, where for all $m, n \in \mathbb{N}$

$$\xi(<m,n>) = \eta(m,n).$$

Here '$<\ ,\ >$' is a primitive recursive surjective pairing function on $\mathbb{N}$, which is expressible in the language of first-order arithmetic, and hence in $\boldsymbol{AssLang}_w$ (*i.e.* the assertion language without arrays; see Zucker [1980], section A.1). More precisely:

THEOREM (Elimination of binary function variables). *For any assertion p containing bound (but not free) occurrences of binary function variables, we can effectively find an assertion $\tilde{p}$ not containing such variables, such that*

$$\mathbb{K} \models p \leftrightarrow \tilde{p}.$$

PROOF. As indicated above, there is an assertion $q_0 \equiv q_0(z_1, z_2, z_3)$ in $\boldsymbol{AssLang}_w$ (with free and bound variables of type N only) which expresses the pairing function $<\ ,\ >$ over $\mathbb{N}$, *i.e.*, for all $k_1, k_2, k_3 \in \mathbb{N}$ and all A, ρ with $\rho(z_i) = k_i$ $(i = 1, 2, 3)$:

$$A, \rho \models q_0(z_1, z_2, z_3) \iff <k_1, k_2> = k_3$$

and $\mathbb{K} \models q_0(z_1, z_2, z_3) \supset ((z_3 \neq \mathsf{unspec}) \leftrightarrow (z_1 \neq \mathsf{unspec} \wedge z_2 \neq \mathsf{unspec}))$.

Now p is transformed into $\tilde{p}$ as follows. Consider any sub-assertion of p of the form $\exists\eta[p_1]$. Suppose η occurs in p_1 in the context of assertion terms $\eta(s_{11}, s_{12}), \ldots, \eta(s_{n1}, s_{n2})$. Then $\exists\eta[p_1]$ is replaced by

$$\exists\xi\, \exists z_1, \ldots, z_n [\bigwedge_{i=1}^{n} q_0(s'_{i1}, s'_{i2}, z_i) \wedge p'_1]$$

where ξ has the same sort as η, and $\xi, z_1, \ldots, z_n$ do not occur, free or bound, in p_1 or q_0, and p'_1, s'_{i1} and s'_{i2} are formed from p_1, s_{i1} and s_{i2} respectively (for $i = 1, \ldots, n$) by replacing all occurrences of $\eta(s_{j1}, s_{j2})$ by $\xi(z_j)$ (for $j = 1, \ldots, n$).

This process is repeated until there are no binary function variables left in p. It should be clear that the resulting assertion $\tilde{p}$ satisfies the statement of the theorem. □

Because of the above theorem, we may assume, when convenient, that the structures A^* include domains $A_i^{*(2)}$ of binary functions, denoted by $\eta, \ldots$.

For such a function η, and $m \in \mathbb{N}$, $\eta(m)$ denotes the unary function $\lambda n\,.\,\eta(m, n)$.

2.6.9 Representation of computation sequences by vectors in A^*

(Compare 2.6.5.) Consider now a vector of elements of A^*:

$$Y = (z^{\mathbf{N}}, \xi_0^{\mathbf{B}}, \xi_1, \ldots, \xi_{M_1}, \eta_1, \ldots, \eta_{M_2}) \qquad (***)$$

where $z^{\mathbf{N}} \in \mathbb{N}$,
$\xi_0^{\mathbf{B}} \in \mathbb{B}^*$,
$\xi_i \in A_{k_i}^*$ for $i = 1, \ldots, M_1$,
and $\eta_j \in A_{l_j}^{*(2)}$ for $j = 1, \ldots, M_2$.

DEFINITIONS. (1) $A^*[\mathbf{N}, \mathbf{B}^*, \vec{k}^*, \vec{l}^{*(2)}]$ is the product domain

$$\mathbb{N}^u \times \mathbb{B}^* \times A_{k_1}^* \times \cdots \times A_{k_{M_1}}^* \times A_{l_1}^{*(2)} \times \cdots \times A_{l_{M_2}}^{*(2)}$$

with typical elements $Y, \ldots$ as in (***).

(2) For $i \in \mathbb{N}$, $Y(i)$ is the "cross-section of Y at i", *i.e.* the following vector in $A^*[\mathbf{B}, \vec{k}, \vec{l}^*]$:

$$(\xi_0^{\mathbf{B}}(i), \xi_1(i), \ldots, \xi_{M_1}(i), \eta_1(i), \ldots, \eta_{M_2}(i)).$$

(3) For a vector Y as in (***), and a finite computation sequence τ over A (see 2.2.5), Y *represents* τ iff $z^{\mathbf{N}} = lh(\tau)$, and for all $i < z^{\mathbf{N}}$, $Y(i)$ represents $\tau(i)$ (as in 2.6.5).

(4) $\ulcorner\tau\urcorner$ is the unique vector in $A^*[\mathsf{N}, \mathsf{B}^*, \vec{k}^*, \vec{l}^{*(2)}]$ which represents τ (assuming τ is finite).

2.6.10 Representation of computation sequences formally in the assertion language

(Compare 2.6.6.) Consider now a tuple of assertion variables

$$Y \equiv (z^{\mathsf{N}}, \xi_0^{\mathsf{B}}, \xi_1, \ldots, \xi_{M_1}, \eta_1, \ldots, \eta_{M_2}) \qquad (****)$$

where ξ_i has type k_i^* for $i = 1, \ldots, M_1$,
and η_j has type $l_j^{*(2)}$ for $j = 1, \ldots, M_2$.

We will use $Y, \ldots$ for tuples of assertion variables as in (****).

Now take $\rho \in \mathrm{VAL}(A^*)$ and a tuple Y as in (****).

DEFINITIONS. (1) $\rho(Y)$ denotes the vector in $A^*[\mathsf{N}, \mathsf{B}^*, \vec{k}^*, \vec{l}^{*(2)}]$:

$$(\rho(z^{\mathsf{N}}), \rho(\xi_0^{\mathsf{B}}), \rho(\xi_1), \ldots, \rho(\xi_{M_1}), \rho(\eta_1), \ldots, \rho(\eta_{M_2})).$$

(2) Y *represents* a finite computation sequence τ relative to ρ iff $\rho(Y)$ represents τ (*i.e.* $\rho(Y) = \ulcorner\tau\urcorner$).

2.6.11 The computation predicate; expressibility of the weakest precondition and strongest postcondition

In this subsection we will define a number of assertions in the language $\boldsymbol{AssLang}_{wa}$, including, notably, the computation predicate Compu_S, and concluding with the assertions $\mathsf{wp}_{wa}[S, p]$ and $\mathsf{sp}_{wa}[p, S]$, representing the weakest precondition and strongest postcondition of S and p.

We first show how various operations on computation sequences (defined in 2.2.5) can be expressed in $\boldsymbol{AssLang}_{wa}$ by corresponding operations on tuples Y. Thus let

$$Y \equiv (z, \xi_0, \xi_1, \ldots, \eta_1, \ldots)$$

as in (****). Then we can define

$$\mathsf{lh}(Y) \equiv_{df} z,$$

$$\mathsf{end}(Y) \equiv_{df} Y(z-1)$$

(note the different typeface from that in 2.2.5). Furthermore, '$Y(k)$', the k-th component of Y, can be defined contextually, thus: an assertion $p(Y(k))$ containing this symbol can be defined as

$$\exists X'[p(X') \wedge X' = Y(k)],$$

where, with $X' \equiv (x_0', x_1', \ldots, \xi_1', \ldots)$, the expression '$X' = Y(k)$' is an abbreviation for

$$\bigwedge_{i=0}^{M_1}(x_i' = \xi_i(k)) \wedge \bigwedge_{j=1}^{M_2}\forall z'[\xi_j'(z') = \eta_j(k, z')].$$

Similarly, the "segmenting operation" '$[Y]_m^n$' can be defined contextually: an assertion $p([Y]_m^n)$ containing this symbol can be defined as

$$\exists Y'[p(Y') \wedge Y' = [Y]_m^n],$$

where the expression '$Y' = [Y]_m^n$' is an abbreviation for

$$\mathsf{lh}(Y') = n - m + 1 \wedge \forall k < \mathsf{lh}(Y')[Y'(k) = Y(m+k)].$$

Similarly for the tail-segment '$[Y]_n$'.

Now we define some assertions containing the tuples $X \equiv (x_0^{\mathsf{B}}, x_1, \ldots, \xi_1, \ldots)$ and Y.

DEFINITION 0. (*a*) $\mathsf{Proper}(X) \equiv_{df} x_0^{\mathsf{B}} = \mathsf{true}$.

(This says: "X represents a proper state".)

(*b*) $\mathsf{Error}(X) \equiv_{df} (x_0^{\mathsf{B}} = \mathsf{false}) \wedge \bigwedge_{i=1}^{M_1}(x_i = \mathsf{unspec}_{k_i}) \wedge \bigwedge_{j=1}^{M_2}\forall z[\xi_j(z) = \mathsf{unspec}_{l_j}]$

("X represents the error state").

(*c*) $\mathsf{State}(X) \equiv_{df} \mathsf{Proper}(X) \vee \mathsf{Error}(X)$.

(*d*) $\mathsf{CompSeq}(Y) \equiv_{df} \forall i < \mathsf{lh}(Y)[\mathsf{Proper}(Y(i))] \wedge \mathsf{State}(\mathsf{end}(Y))$

("Y represents a finite computation sequence").

Next, the notion of *variant* of a tuple representing a state is defined contextually:

DEFINITION 1 (*Variants* of tuples representing states).

Suppose $X \equiv (x_0, x_1, \ldots, x_{M_1}, \xi_1, \ldots, \xi_{M_2})$

and $X' \equiv (x_0', x_1', \ldots, x_{M_1}', \xi_1', \ldots, \xi_{M_2}')$

are disjoint tuples. Then

(*a*) '$X' = X\{s/i_0\}$' (for $1 \leq i_0 \leq M_1$ and s an assertion term of sort k_{i_0}) is the assertion:

$$(x_{i_0}' = s) \wedge \bigwedge_{\substack{i=0 \\ i \neq i_0}}^{M_1}(x_i' = x_i) \wedge \bigwedge_{j=1}^{M_2}\forall z[\xi_j'(z) = \xi_j(z)].$$

(*b*) '$X' = X\{s/<j_0, s_0>\}$ (for $1 \le j_0 \le M_2$, s of sort l_{j_0} and s_0 of sort N) is the assertion:

$$\bigwedge_{i=0}^{M_1}(x_i' = x_i) \wedge \xi_{j_0}(s_0) = s \wedge \forall z \neq s_0[\xi_{j_0}'(z) = \xi_{j_0}(z)] \wedge$$
$$\wedge \bigwedge_{\substack{j=1 \\ j \neq j_0}}^{M_2} \forall z\,[\xi_j'(z) = \xi_j(z)].$$

We now confirm the correctness of this definition.

THEOREM 1. *Suppose* X *represents* $\sigma(\neq \varepsilon)$ *relative to* ρ, *and* $\mathbf{ProgVar}(s, s_0) = \emptyset$. *Then*

(*a*) $A^*, \rho \vDash X' = X\{s/i\} \Leftrightarrow$

X' *represents* $\sigma\{\mathcal{S}_A(s)(\rho, \cdot)/v_i\}$ *relative to* ρ.

(*b*) $A^*, \rho \vDash X' = X\{s/<j, s_0>\} \Leftrightarrow$

X' *represents* $\sigma\{\mathcal{S}_A(s)(\rho, \cdot)/<a_j, \mathcal{S}_A(s_0)(\rho, \cdot)>\}$ *relative to* ρ.

PROOF. Directly from the definitions. □

DEFINITION 2 (*The computation predicate*).
We will define an assertion $\mathsf{Compu}_S(X, Y, X')$ with the meaning: "Y represents a computation sequence generated by S, starting in the state represented by X and terminating in the state represented by X'" (see Theorem 2 below).

The definition is by induction on $\mathbf{compl}(S)$. (Compare the definition of $\mathbf{Comp}_A$ in 2.2.6. An explanation follows step (6) below.)

(1) $S \equiv \mathsf{skip}$. Then $\mathsf{Compu}_S(X, Y, X') \equiv$

$$\mathsf{lh}(Y) = 2 \wedge Y(0) = Y(1) = X = X' \wedge \mathsf{Proper}(X).$$

(2) $S \equiv v_i := t \quad (1 \le i \le M_1)$. Then $\mathsf{Compu}_S(X, Y, X') \equiv$

$$\mathsf{lh}(Y) = 2 \wedge Y(0) = X \wedge Y(1) = X' \wedge \mathsf{Proper}(X) \wedge$$
$$\wedge (t\,[\![X]\!] \neq \mathsf{unspec} \supset X' = X\{t\,[\![X]\!]/i\}) \wedge$$
$$\wedge (t\,[\![X]\!] = \mathsf{unspec} \supset \mathsf{Error}(X')]).$$

(3) $S \equiv a_j[t_0] := t \quad (1 \le j \le M_2)$. Then $\mathsf{Compu}_S(X, Y, X') \equiv$

$$\mathsf{lh}(Y) = 2 \wedge Y(0) = X \wedge Y(1) = X' \wedge \mathsf{Proper}(X) \wedge$$
$$\wedge ((t_0[\![X]\!] \neq \mathsf{unspec} \wedge t[\![X]\!] \neq \mathsf{unspec}) \supset$$
$$\supset X' = X\{t[\![X]\!]/\langle j, t_0[\![X]\!]\rangle\}) \wedge$$
$$\wedge ((t_0[\![X]\!] = \mathsf{unspec} \vee t[\![X]\!] = \mathsf{unspec}) \supset \mathsf{Error}(X)).$$

(4) $S \equiv S_1; S_2$. Then $\mathsf{Compu}_S(X, Y, X') \equiv$

$$\exists m < \mathsf{lh}(Y) \exists X''[\mathsf{Proper}(X'') \wedge \mathsf{Compu}_{S1}(X, [Y]_0^m, X'') \wedge$$
$$\wedge \mathsf{Compu}_{S2}(X'', [Y]_{m+1}, X')] \vee$$
$$\vee (\mathsf{Compu}_{S1}(X, Y, X') \wedge \mathsf{Error}(X')).$$

(5) $S \equiv \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$. Then $\mathsf{Compu}_S(X, Y, X') \equiv$

$$Y(0) = X \wedge$$
$$\wedge (b[\![X]\!] = \mathsf{true} \supset \mathsf{Compu}_{S1}(X, [Y]_1, X')) \wedge$$
$$\wedge (b[\![X]\!] = \mathsf{false} \supset \mathsf{Compu}_{S2}(Y(1), [Y]_1, X')) \wedge$$
$$\wedge (b[\![X]\!] = \mathsf{unspec} \supset (\mathsf{lh}(Y) = 2 \wedge Y(1) = X' \wedge \mathsf{Error}(X'))).$$

(6) $S \equiv \mathsf{while}\ b\ \mathsf{do}\ S_0\ \mathsf{od}$. Then $\mathsf{Compu}_S(X, Y, X') \equiv$

$$Y(0) = X \wedge$$
$$\wedge \exists m, n, \zeta[\zeta(0) = 0 \wedge \zeta(n) = m \wedge \forall i < n[\zeta(i) < \zeta(i+1)] \wedge$$
$$\wedge \forall i < n[b[\![Y(\zeta(i))]\!] = \mathsf{true} \wedge$$
$$\wedge \mathsf{Compu}_{S0}(Y(\zeta(i)), [Y]_{\zeta(i)+1}^{\zeta(i+1)}, Y(\zeta(i+1)))] \wedge$$
$$\wedge (\mathsf{case1} \vee \mathsf{case2} \vee \mathsf{case3})].$$

where case1, case2 and case3 are, respectively, the three assertions

$$b[\![Y(m)]\!] = \mathsf{false} \wedge Y(m+1) = Y(m) = X' \wedge \mathsf{lh}(Y) = m+2,$$
$$b[\![Y(m)]\!] = \mathsf{unspec} \wedge Y(m+1) = X' \wedge \mathsf{Error}(X') \wedge \mathsf{lh}(Y) = m+2,$$
$$b[\![Y(m)]\!] = \mathsf{true} \wedge \mathsf{Compu}_{S0}(Y(m), [Y]_{m+1}, X') \wedge \mathsf{Error}(X').$$

Explanation. Here m and n range over type $\mathbb{N}$, and ζ over $\mathbb{N}^*$ (see 2.3.4). The sequence of numbers $\zeta(0), \zeta(1), \ldots, \zeta(n)$ marks off the n segments of the computation sequence Y determined by n repeated executions of the 'while' loop. The three assertions case1, case2 and case3 give the three

ways that a **'while'** statement can terminate (corresponding to the first three cases listed under (4) in 2.2.6).

REMARKS. (1) It is easily seen (by induction on $\boldsymbol{compl}(s)$) that

$$\mathbb{K} \models \mathsf{Compu}_S(X,Y,X') \supset (\mathsf{CompSeq}(Y) \wedge Y(0)=X \wedge \mathsf{end}(Y)=X')$$

(2) It is also easily seen that

$$\boldsymbol{ProgVar}(\mathsf{Compu}_S(X,Y,X')) = \emptyset,$$

$$\boldsymbol{AssVar}(\mathsf{Compu}_S(X,Y,X')) \subseteq X,Y,X'.$$

(Hence we may use the notation of 1.3.12, Remark (1).)

Again, we confirm the correctness of the definition.

THEOREM 2. *The assertion* Compu_S *expresses the computation predicate uniformly over* $\mathbb{K}$. *In other words: if* X, Y *and* X' *represent* σ, τ *and* σ' *respectively (all relative to* ρ*), then*

$$A^*, \rho \models \mathsf{Compu}_S(X,Y,X') \Leftrightarrow$$

$$\sigma \neq \varepsilon \text{ and } \tau = \boldsymbol{Comp}_A(S)(\sigma) \text{ and } \sigma' = \boldsymbol{end}(\tau) = \mathcal{M}_A(S)(\sigma).$$

PROOF. Induction on $\boldsymbol{compl}(S)$. □

As we have observed before (2.6.3), this theorem provides the justification for the weak second order assertion language, since it shows that the semantics of all **'while'** program computations can be expressed uniformly (for a given S) in this language.

The assertion Compu_S may be compared to (the assertion representing) Kleene's T-predicate (Kleene [1952], Chapter XI), which is first-order definable over the structure of Peano arithmetic. Since Compu_S is not first-order definable over the signature Σ of A, the extension to Σ^* is needed.

DEFINITION 3. $\mathsf{Val}(X,\vec{v},\vec{a}) \equiv_{df}$

$$\mathsf{Proper}(X) \wedge \bigwedge_{i=1}^{M_1} x_i = v_i \wedge \bigwedge_{j=1}^{M_2} \forall z[\xi_j(z)=a_j[z]].$$

(This says: "X is a proper state, with values given by $\vec{v}$ and $\vec{a}$.")

THEOREM 3. *For any* ρ *and* $\sigma \neq \varepsilon$*:*

$$A^*, \rho, \sigma \models \mathsf{Val}(X,\vec{v},\vec{a}) \Leftrightarrow$$

$$X \textit{ represents } \sigma \textit{ relative to } \rho \textit{ (i.e. } \rho(X) = \ulcorner \sigma \urcorner).$$

PROOF. Clear. □

The following corollary of Theorem 3 will only be used in Chapter 3.

COROLLARY. *Suppose* $\boldsymbol{Var}(p) \cap X = \emptyset$. *Then*

$$\mathbb{K} \vDash p \leftrightarrow \exists X\,[\mathsf{Val}(X, \vec{v}, \vec{a}) \wedge p[\![X]\!]].$$

PROOF. We will show that for any interpretation $I = (A^*, \rho, \sigma)$,

$$I \vDash p \iff I \vDash \exists X\,[\mathsf{Val}(X, \vec{v}, \vec{a}) \wedge p[\![X]\!]].$$

("$\Rightarrow$") Suppose (dropping the 'A^*')

$$\rho, \sigma \vDash p\,. \tag{1}$$

Let $\overline{\rho} = \rho\{\ulcorner\sigma\urcorner / X\}$. (See 2.6.5, Definition 4, and 2.6.6, Definition 3.) Then X represents σ relative to $\overline{\rho}$, so by Theorem 3

$$\overline{\rho}, \sigma \vDash \mathsf{Val}(X, \vec{v}, \vec{a})\,. \tag{2}$$

Also, by (1) and since $\boldsymbol{Var}(p) \cap X = \emptyset$,

$$\overline{\rho}, \sigma \vDash p$$

(by Theorem 2 of 1.3.12, which applies to the present assertion language). Hence by the Theorem in 2.6.7

$$\overline{\rho}, \sigma \vDash p[\![X]\!]. \tag{3}$$

By (2) and (3), $\quad \overline{\rho}, \sigma \vDash \mathsf{Val}(X, \vec{v}, \vec{a}) \wedge p[\![X]\!]$.

Hence $\quad \rho, \sigma \vDash \exists X\,[\mathsf{Val}(X, \vec{v}, \vec{a}) \wedge p[\![X]\!]]. \tag{4}$

("$\Leftarrow$") Conversely, suppose (4). Then for some vector $\boldsymbol{X}$, putting $\overline{\rho} = \rho\{\boldsymbol{X} / X\}$:

$$\overline{\rho}, \sigma \vDash \mathsf{Val}(X, \vec{v}, \vec{a}) \tag{5}$$

and

$$\overline{\rho}, \sigma \vDash p[\![X]\!]. \tag{6}$$

By (5) and Theorem 3, X represents σ relative to $\overline{\rho}$, hence by (6) and the Theorem in 2.6.7,

$$\overline{\rho}, \sigma \vDash p\,.$$

Hence, since $\boldsymbol{Var}(p) \cap X = \emptyset$, we have (again by 1.3.12, Theorem 2):

$$\rho, \sigma \vDash p\,. \qquad \square$$

Now we can give the assertion expressing the *weakest precondition* of any statement S and assertion p (satisfying the convention in 2.6.4):

DEFINITION 4. $\mathsf{wp}_{wa}[S,p] \equiv_{df}$

$$\forall X,Y,X'[\mathsf{Val}(X,\vec{v},\vec{a}) \wedge \mathsf{Compu}_S(X,Y,X') \supset$$
$$\supset \mathsf{Proper}(X') \wedge p[\![X]\!]].$$

(Here X' is chosen so that $\boldsymbol{Var}(p) \cap X' = \emptyset$.)

We note that $\boldsymbol{ProgVar}(\mathsf{wp}_{wa}[S,p]) = \vec{v},\vec{a}$,

and $\boldsymbol{AssVar}(\mathsf{wp}_{wa}[S,p]) = \boldsymbol{AssVar}(p)$.

(Below we will drop the subscript '*wa*.')

THEOREM 4. $\mathsf{wp}[S,p]$ *expresses the weakest precondition of* S *and* p, *uniformly over* $\mathbb{K}$. *In other words, for all interpretations* (A^*,ρ,σ) *in* $\mathbb{K}$,

$$A^*,\rho,\sigma \models \mathsf{wp}[S,p] \iff$$

for all σ', *if* $\mathfrak{M}_A(S)(\sigma)\downarrow\sigma'$ *then* $\sigma' \neq \varepsilon$ *and* $A^*,\rho,\sigma' \models p$.

PROOF. Clear, from Theorems 2 and 3 and the Theorem in 2.6.7. □

COROLLARIES. (Compare 1.6.6.)

(1) $\mathbb{K} \models \{q\}S\{p\} \iff \mathbb{K} \models q \supset \mathsf{wp}[S,p]$.

In particular, taking $q \equiv \mathsf{wp}[S,p]$:

$$\mathbb{K} \models \{\mathsf{wp}[S,p]\}S\{p\}.$$

(2) (Intermediate assertion.)

$$\mathbb{K} \models \{q\}S_1;S_2\{p\} \iff \mathbb{K} \models \{q\}S_1\{\mathsf{wp}[S_2,p]\}.$$

Finally, we have the assertion expressing the *strongest postcondition.*

DEFINITION 5. $\mathsf{sp}_{wa}[p,S] \equiv_{df}$

$$\exists X',Y,X[p[\![X']\!] \wedge \mathsf{Compu}_S(X',Y,X) \wedge \mathsf{Val}(X,\vec{v},\vec{a})].$$

(Again, X' is chosen so that $\boldsymbol{Var}(p) \cap X' = \emptyset$.)

We note again that $\boldsymbol{ProgVar}(\mathsf{sp}_{wa}[p,S]) = \vec{v},\vec{a}$,

and $\boldsymbol{AssVar}(\mathsf{sp}_{wa}[p,S]) = \boldsymbol{AssVar}(p)$.

(As before, we will drop the subscript '*wa*'.)

THEOREM 5. $\mathsf{sp}[p,S]$ *expresses the strongest postcondition of* S *and* p, *uniformly over* $\mathbb{K}$. *In other words, for all interpretations* (A^*, ρ, σ) *in* $\mathbb{K}$,

$$A^*, \rho, \sigma \models \mathsf{sp}[p,S] \iff$$

$$\text{for some } \sigma' \neq \varepsilon,\ A^*, \rho, \sigma' \models p \text{ and } \mathcal{M}_A(S)(\sigma') \downarrow \sigma.$$

COROLLARIES. (Compare 1.6.8.)

(1) $\mathbb{K} \models \{p\}S\{q\} \Rightarrow (\mathbb{K} \models \{p\}S\{\mathsf{sp}[p,S]\}$ and $\mathbb{K} \models \mathsf{sp}[p,S] \supset q)$.

(2) (Intermediate assertion).

$\mathbb{K} \models \{p\}S_1;S_2\{q\} \Rightarrow (\mathbb{K} \models \{p\}S_1\{\mathsf{sp}[p,S_1]\}$ and $\mathbb{K} \models \{\mathsf{sp}[p,S_1]\}S_2\{q\})$.

2.7 COMPLETENESS OF THE PROOF SYSTEM

2.7.1 Proof of completeness

THEOREM. *The system* $\mathbf{ProofSys}_{wa}(\mathbb{K})$ *is complete relative to* $\mathbb{K}$, *i.e. for any* $f \in \mathbf{Form}_{wa}(\Sigma)$,

$$\mathbb{K} \models f \Rightarrow \mathbb{K} \vdash f. \tag{1}$$

PROOF. We prove (1) for $f \equiv \{p\}S\{q\}$, by induction on $\mathbf{compl}(S)$. Most of the cases are as in the proof of the Completeness Theorem in Chapter 1 (1.7.1). For example, in the case $S \equiv S_1;S_2$, an intermediate assertion can be defined as either $\mathsf{wp}[S_2,q]$ or $\mathsf{sp}[p,S_1]$. Consider now the case

$$S \equiv \mathsf{while}\ b\ \mathsf{do}\ S_0\ \mathsf{od}. \tag{2}$$

So suppose

$$\mathbb{K} \models \{p\}S\{q\} \tag{3}$$

with S as in (2). Let

$$r \equiv \mathsf{wp}[S,q].$$

By (3) and Corollary (1) of Theorem 4 (in 2.6.11)

$$\mathbb{K} \models p \supset r. \tag{4}$$

Also, by the same Corollary,

$$\mathbb{K} \models \{r\}S\{q\}. \tag{5}$$

Now $\mathcal{M}_A(S) = \mathcal{M}_A(\mathsf{if}\ b\ \mathsf{then}\ S_0\ \mathsf{fi};S)$ (*check!*); so from (5)

$$\mathbb{K} \models \{r\}\mathsf{if}\ b\ \mathsf{then}\ S_0\ \mathsf{fi};S\{q\}.$$

Hence from Corollary (2) of Theorem 4 (in 2.6.11),

$$\mathbb{K} \models \{r\}\mathsf{if}\ b\ \mathsf{then}\ S_0\ \mathsf{fi}\{r\}. \tag{6}$$

Hence $\mathbb{K} \models \{r \wedge (b = \mathbf{true})\} S_0 \{r\}$

and so, by induction hypothesis,

$$\mathbb{K} \vdash \{r \wedge (b = \mathbf{true})\} S_0 \{r\}. \tag{7}$$

Also $\mathcal{R}_A(b)(\sigma) = \mathbb{t} \Rightarrow \mathcal{M}_A(S) = \mathbf{skip}$,

so from (5) $\mathbb{K} \models r \wedge (b = \mathbf{false}) \supset q$ (8)

Also from (6), $\mathbb{K} \models r \supset (b \neq \mathbf{unspec})$. (9)

Thus the premisses of the **'while'** rule have been proved, by (4), (7), (8), (9) and the oracle rule. Hence we can infer the conclusion, *i.e.* $\mathbb{K} \models \{p\} S \{q\}$. □

2.7.2 Remark on the loop invariant

In the above proof, the assertion expressing the *weakest precondition* was used to obtain a loop invariant for the **'while'** statement. Such a loop invariant could also have been obtained by an assertion expressing an appropriate *strongest postcondition* (as was first done in Cook [1976]: see Apt [1981], §2.8).

2.8 APPENDIX: TOTAL CORRECTNESS FOR 'WHILE' PROGRAMS

2.8.1 Semantics of total correctness

As indicated in 2.4.3, the satisfaction relation '$I \models \{p\} S \{q\}$' defined in 2.4.2 does not require termination of the program S, only that *if* S terminates, then it does so in a proper (non-error) state which satisfies q.

This notion of *partial correctness* is the one that is emphasized in this monograph.

If, however, we want to incorporate the assumption of termination in the definition of satisfaction, then we arrive at the notion of *total correctness*. In this concluding section to Chapter 2 we will briefly re-work some of the results of this chapter in terms of this notion. (See also Apt [1981], §2.11.)

First we define, for a correctness formula f and interpretation $I = (A^*, \rho, \sigma)$, the notion $I \models^T f$, *I satisfies f totally*, or f is *totally correct under I*, as follows. If f is an assertion, then $I \models^T f$ iff $I \models f$ as before. If $f \equiv \{p\} S \{q\}$ then $I \models^T f$ iff (*cf.* 2.4.2, Definition 1)

$$A^*, \rho, \sigma \models p \Rightarrow$$
$$\text{for some } \sigma' \neq \varepsilon,\ \mathcal{M}_A(S)(\sigma) \downarrow \sigma' \text{ and } A^*, \rho, \sigma' \models q\ .$$

Total $\mathbb{K}$-validity is then defined, as before, by:

$$\mathbb{K} \models^T f \quad \textit{iff} \quad I \models^T f \text{ for all } I \in \mathrm{INTERP}^*(\mathbb{K}).$$

REMARKS. (1) The notion of satisfaction '$I \models f$' defined previously (2.4.2, Definition 1) can then be stated as: *I satisfies f partially* or f is *partially correct* under I.

(2) Now there is *not* the sharp distinction between non-termination of a program and abortion (as was the case with partial correctness; see 2.4.3). Thus (to return to the trivial example in 2.4.3) for $I = (A^*, \rho, \sigma)$, $I \not\models^T \{\mathsf{true}\} S \{q\}$ if *either* $\mathcal{M}_A(S)(\sigma) \uparrow$ *or* $\mathcal{M}_A(S)(\sigma) \downarrow \varepsilon$.

2.8.2 A proof system

A proof system $\textbf{\textit{ProofSys}}^T_{wa} = \textbf{\textit{ProofSys}}^T_{wa}(\mathbb{K})$ for total correctness of 'while' programs can be obtained from $\textbf{\textit{ProofSys}}_{wa}$ (2.5.1) by replacing the 'while' rule (A.5) by the following:

(A.5)^T *The* 'while' *rule for total correctness*:

$$\frac{p \supset \exists n\,[r(n)],\quad r(n+1) \supset (b = \mathsf{true}),\quad \{r(n+1)\} S \{r(n)\},\quad r(0) \supset (b = \mathsf{false}) \wedge q}{\{p\}\,\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}\{q\}}$$

where $r(z)$ is an assertion with the free assertion variable z of type N, not free in p or q, and n ranges over natural numbers, thus (recall the notation of 2.3.4):

$$\begin{aligned} \exists n\,[r(n)] \quad &\text{means} \quad \exists z\,[z \neq \mathsf{unspec}_{\mathsf{N}} \wedge r(z)], \\ r(n+1) \supset (b = \mathsf{true}) \quad &\text{means} \quad (z \neq \mathsf{unspec}_{\mathsf{N}} \wedge r(z)) \supset (b = \mathsf{true}), \\ \{r(n+1)\} S \{r(n)\} \quad &\text{means} \quad \{z \neq \mathsf{unspec}_{\mathsf{N}} \wedge r(z+1)\} S \{r(z)\}. \end{aligned}$$

This rule was formulated by Harel (in the context of "arithmetical universes": see Harel [1979], §3.3). The assertion $r(z)$ is called a *convergent* for the 'while' statement S (with respect to p and q), or a "loop convergent". Intuitively, $r(n)$ says that after n more executions of the 'while' loop, the value of b will change to $\mathbb{f}$, thus terminating the loop (in the state q).

The system $\textbf{\textit{ProofSys}}^T_{wa}$ is sound and complete for total correctness. The proof of *soundness* is fairly routine. For *completeness*, we will again consider the notions of weakest precondition and strongest postcondition, in the next subsection.

2.8.3 Weakest precondition and strongest postcondition for total correctness

We define (*cf.* 2.6.2, Definition 1), for a state transformer Φ on A and a predicate π on A, the *weakest precondition for total correctness* of Φ and π, written $\mathrm{WP}^T_A(\Phi, \pi)$, as the predicate on A which holds at a proper state σ if, and only if,

$$\Phi(\sigma)\downarrow \neq \varepsilon \quad \text{and} \quad \pi(\Phi(\sigma)) = \mathrm{tt}.$$

Next, the weakest precondition for total correctness is $\mathbb{K}$-expressible by the assertion (*cf.* $\mathsf{wp}_{wa}[S,p]$ in 2.6.11, Definition 4): $\mathsf{wp}^T_{wa}[S,p] \equiv_{df}$

$$\exists X, Y, X'[\mathsf{Val}(X,\vec{v},\vec{a}) \wedge \mathsf{Compu}_S(X,Y,X') \wedge \mathsf{Proper}(X') \wedge p[\![X']\!]].$$

This satisfies (*cf.* 2.6.11, Theorem 4)

$$A^*, \rho, \sigma \models^T \mathsf{wp}^T[S,p] \iff$$

$$\textit{for some } \sigma',\ \mathcal{M}_A(S)(\sigma)\downarrow \sigma' \neq \varepsilon \textit{ and } A^*, \rho, \sigma' \models^T p\,.$$

As corollaries we have (*cf.* 2.6.11, Theorem 4, Corollaries)

(1) $\mathbb{K} \models^T \{q\}S\{p\} \iff \mathbb{K} \models^T q \supset \mathsf{wp}^T[S,p]$.

In particular, taking $q \equiv \mathsf{wp}^T[S,p]$:

$$\mathbb{K} \models^T \{\mathsf{wp}^T[S,p]\}S\{p\}.$$

(2) (Intermediate assertion.)

$$\mathbb{K} \models^T \{q\}S_1;S_2\{p\} \iff \mathbb{K} \models^T \{q\}S_1\{\mathsf{wp}^T[S_2,p]\}.$$

The *strongest postcondition for total correctness* is defined *exactly* as in 2.6.2, Definition (2), and is expressible by the assertion $\mathsf{sp}_{wa}[p,S]$ (2.6.11, Definition 5), since this assertion also satisfies the versions of Theorem 5 and its Corollaries (2.6.11) formed when '$\models$' is replaced throughout by '$\models^T$'.

2.8.4 Proof of completeness

The completeness theorem for $\boldsymbol{ProofSys}^T_{wa}$, for correctness formulas $\{p\}S\{q\}$, is proved, as before, by induction on $\boldsymbol{compl}(S)$ (*cf.* 2.7.1).

In the case $S \equiv S_1;S_2$, an intermediate assertion may be defined as $\mathsf{wp}^T[S_2,q]$, *or* as $\mathsf{wp}[S_2,q]$, *or* as $\mathsf{sp}[p,S_1]$!

Consider now the case $S \equiv \mathsf{while}\ b\ \mathsf{do}\ S_0\ \mathsf{od}$, and suppose $\models^T \{p\}S\{q\}$. We can define a loop convergent $r(z)$ which asserts that execution of S consists of exactly z iterations of the **'while'** loop, following which b is false and q holds. To be precise, $r(z)$ is the assertion

$$\exists X, Y, X'[\mathsf{Val}(X,\vec{v},\vec{a}) \wedge Y(0)=X \wedge \mathsf{end}(Y)=X' \wedge$$
$$\wedge \exists m, \zeta[\zeta(0)=0 \wedge \zeta(z)=m \wedge \forall i<z[\zeta(i)<\zeta(i+1)] \wedge$$
$$\wedge \forall i<z[b[\![Y(\zeta(i))]\!]=\mathsf{true} \wedge$$
$$\wedge \mathsf{Compu}_{S0}(Y(\zeta(i)), [Y]^{\zeta(i+1)}_{\zeta(i)+1}, Y(\zeta(i+1)))] \wedge$$
$$\wedge \mathsf{case1} \wedge$$
$$\wedge q[\![Y(m)]\!]].$$

(*Cf.* 2.6.11, Definition 2, part (6). We assume that z is not in X or Y or X'.) Now, using this assertion $r(z)$ as a loop convergent, we can deduce the validity of the assumptions of the '**while**' rule $(A.5)^T$, etc.

2.8.5 Historical remark

The first proof rule for total correctness of '**while**' programs, within the framework of Hoare logic, was given by Manna and Pnueli (1974).

Proof systems for partial and total correctness of '**while**' programs for abstract structures, together with completeness proofs, were given by Harel (1979) (in the context of dynamic logic). The '**while**' rule for total correctness in 2.8.2 is a version of Harel's rule.

Note that in order to prove expressibility of the computation predicate, and hence of the loop invariant (in the case of partial correctness), or loop convergent (in the case of total correctness), we need to be able to speak, in the assertion language, about (1) the natural numbers with their standard operations, and (2) arbitrary finite sequences of elements of the abstract structure.

Of course, the natural numbers are needed even to state the '**while**' rule for total correctness.

Harel deals with this by considering (single-sorted) *arithmetical structures*, which are (following Parikh) structures which (1) contain the natural numbers, with their standard operations, as a definable substructure, and (2) admit an encoding of finite sequences of elements of the domain into single elements.

As Harel points out, any (single-sorted) structure can be extended to a (single-sorted) arithmetical structure by augmenting it, if necessary, with the natural numbers, and additional apparatus for encoding finite sequences. However the imposition of an encoding of finite sequences on an abstract structure is, in general, algebraically unsatisfactory.

Our approach (as we have shown) is (1) to work with many-sorted structures A, with $\mathbf{N}$ as one of the sorts, and (2) to express finite sequences over A by means of the structure A^*. This seems more satisfactory from a computational point of view.

Chapter 3

Recursive Programs

3.1 THE PROGRAMMING LANGUAGE

3.1.1 Syntax

The programming language $\boldsymbol{ProgLang}_{ra} = \boldsymbol{ProgLang}_{ra}(\Sigma)$ ('*ra*' for "recursive with arrays") is like the language $\boldsymbol{ProgLang}_{sa}(\Sigma)$ of Chapter 1, except that there are also (parameterless) *procedure variables*, and *declarations* of these.

Thus we have the classes:

(1) ***ProgVar*** and ***ProgTerm*** as before;

(2) ***ProcVar***, the class of *procedure variables*, denoted $P, \ldots$;

(3) $\boldsymbol{Statemt}_{ra}$, the class of *statements*, denoted $S, \ldots$, and defined by

$$S ::= \mathsf{skip} \mid v^i := t^i \mid a^i[t^N] := t^i \mid S_1; S_2 \mid \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid P;$$

(4) ***Decl***, the class of declarations, denoted $D, \ldots$, and defined by

$$D ::= P_1 \Leftarrow S_1, \ldots, P_m \Leftarrow S_m \qquad (m \geq 0),$$

also written $< P_i \Leftarrow S_i >_{i=1}^{m}$, where $P_i \not\equiv P_j$ for $i \neq j$;

(5) ***Prog***, the class of *recursive programs*, denoted $R, \ldots$, and defined by

$$R ::= < D : S >$$

which we will write as $< D \mid S >$ or $<< P_i \Leftarrow S_i >_{i=1}^{m} \mid S >$.

(Of course, these classes are all defined relative to the signature Σ; thus, *e.g.*, $\boldsymbol{Statemt}_{ra}(\Sigma)$, $\boldsymbol{Prog}(\Sigma)$.)

3.1.2 The programming language without arrays

Again, there is a version of the language without arrays, $\boldsymbol{ProgLang}_r(\Sigma)$.

3.1.3 Further definitions

DEFINITION 1. (*a*) $\boldsymbol{ProgVar}(S)$ is, as before, the set of program variables in S.

(*b*) For $D \equiv < P_i \Leftarrow S_i >_{i=1}^m$, we define

$$\boldsymbol{ProgVar}(D) \equiv_{df} \bigcup_{i=1}^{m} \boldsymbol{ProgVar}(S_i).$$

(*c*) For $R \equiv < D \mid S >$, we define

$$\boldsymbol{ProgVar}(R) \equiv_{df} \boldsymbol{ProgVar}(D) \cup \boldsymbol{ProgVar}(S).$$

DEFINITION 2. (*a*) A declaration $< P_i \Leftarrow S_i >_{i=1}^m$ is *closed* if all the procedure variables occurring in any of the S_i are among $P_1, \ldots, P_m$.

(*b*) A program $<< P_i \Leftarrow S_i >_{i=1}^m \mid S >$ is *closed* if all the procedure variables occurring in S or any of the S_i are among $P_1, \ldots, P_m$.

(*c*) $\boldsymbol{ClProg}$ is the class of *closed programs.*

3.1.4 Closed declarations and programs: Convention

From now on, we will only consider *closed declarations and programs.* So in fact $D, \ldots$ will range over closed declarations, and $R, \ldots$ over closed programs.

3.1.5 Operational semantics

The set COMPSEQ(A) of *computation sequences* over A is defined as in Chapter 2 (see 2.2.5).

We will again want to define a function

$$\boldsymbol{Comp}_A : \boldsymbol{ClProg} \rightarrow (\text{PR.STATE}(A) \rightarrow \text{COMPSEQ}(A))$$

so that $\boldsymbol{Comp}_A(R)(\sigma)$ is the computation sequence generated by R, starting in state $\sigma(\neq \varepsilon)$.

Suppose we try to define $\boldsymbol{Comp}_A(R)$, with $R \equiv < D \mid S >$, by induction on $\boldsymbol{compl}(S)$ (as in 2.2.6). Now the cases that S is a **skip**, assignment, composition or conditional statement can be treated as before; but there is a new case, namely that S is a procedure variable, say $S \equiv P_i$ for some i, $1 \leq i \leq m$, where $D \equiv < P_j \Leftarrow S_j >_{j=1}^m$. Then

$$\mathbf{Comp}_A(\langle D \mid P_i \rangle)(\sigma) = (\sigma)^\frown \mathbf{Comp}_A(\langle D \mid S_i \rangle)(\sigma).$$

However, such a "definition" is problematical as it stands. It is *not* simply an induction on $\mathbf{compl}(S)$, since (taking the above case) S_i is more complex than P_i.

So instead, following De Bruin [1984], we proceed as follows. We define a function

$$\mathbf{CompStep}_A : \mathbb{N} \to (\mathbf{ClProg} \to (\mathrm{PR.STATE}(A) \to \mathrm{STATE}(A) \cup \{*\}))$$

where '$*$' is a new symbol or object, the idea being that $\mathbf{CompStep}_A(n)(R)$ is the nth *step of the computation sequence.* That is, if $\tau = \mathbf{Comp}_A(R)(\sigma)$ with $\mathbf{lh}(\tau) = l$, then

$$\mathbf{CompStep}_A(n)(R)(\sigma) = \begin{cases} \tau(n) & \text{for } n < l \\ * & \text{for } n \geq l. \end{cases}$$

The definition of $\mathbf{CompStep}_A(n)(D \mid S)(\sigma)$ is by induction on the pair $(n, \mathbf{compl}(S))$, *i.e.* a main induction on n, and a secondary induction on $\mathbf{compl}(S)$.

(Notice that we write '$D \mid S$' instead of '$\langle D \mid S \rangle$' in the context '$\mathbf{CompStep}_A(n)(\ \ )$', and likewise in other such contexts below.)

First, for $n = 0$, and any S:

$$\mathbf{CompStep}_A(0)(D \mid S)(\sigma) = \sigma.$$

Next, for $n > 0$, consider the various cases for S.

Case 1. S is skip or an assignment.

Case 1(a). $n = 1$.

$$\mathbf{CompStep}_A(1)(D \mid S)(\sigma) = \mathcal{M}_A(S)(\sigma)$$

where $\mathcal{M}_A$ is as defined in 1.2.11.

Case 1(b). $n > 1$.

$$\mathbf{CompStep}_A(n)(D \mid S)(\sigma) = *.$$

Case 2. $S \equiv S_1; S_2$.

$$\mathbf{CompStep}_A(n)(D \mid S)(\sigma) = \begin{cases} \mathbf{CompStep}_A(n)(D \mid S_1)(\sigma) \\ \qquad \text{if } \mathbf{CompStep}_A(n)(D \mid S_1) \neq * \\ \mathbf{CompStep}_A(n-\tilde{l})(D \mid S_2)(\tilde{\sigma}) \\ \qquad \text{otherwise, if } \tilde{\sigma} \neq \varepsilon \text{ (see below)} \\ * \qquad \text{otherwise} \end{cases}$$

where $\tilde{l}$ = least l s.t. $\mathbf{CompStep}_A(l)(D \mid S_1)(\sigma) = *$

and $\tilde{\sigma} = \mathbf{CompStep}_A(\tilde{l}-1)(D \mid S_1)(\sigma)$.

(Think of $\tilde{l}$ as $\mathbf{lh}(\mathbf{Comp}_A(D \mid S_1)(\sigma))$, and $\tilde{\sigma}$ as $\mathbf{end}(\mathbf{Comp}_A(D \mid S_1)(\sigma))$.)

Case 3. $S \equiv$ **if** b **then** S_1 **else** S_2 **fi**.

$$\mathbf{CompStep}_A(n)(D \mid S)(\sigma) = \begin{cases} \mathbf{CompStep}_A(n-1)(D \mid S_1)(\sigma) \\ \qquad \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{t}, \\ \mathbf{CompStep}_A(n-1)(D \mid S_2)(\sigma) \\ \qquad \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{f}, \\ \varepsilon \qquad \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{u} \text{ and } n = 1, \\ * \qquad \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{u} \text{ and } n > 1. \end{cases}$$

Case 4. $S \equiv P_i$ $(1 \leq i \leq m$, where $D \equiv < P_j \Leftarrow S_j >_{j=1}^{m})$.

This is the most interesting case:

$$\mathbf{CompStep}_A(n)(D \mid P_i)(\sigma) = \mathbf{CompStep}_A(n-1)(D \mid S_i)(\sigma).$$

This completes the definition of $\mathbf{CompStep}_A$.

Now we can define the *length of a computation sequence* directly from $\mathbf{CompStep}_A$, as the function

$$\mathbf{LengthComp}_A : \mathbf{ClProg} \to (\mathrm{PR.STATE}(A) \to \mathbb{N} \cup \{\infty\})$$

where

$$\mathbf{LengthComp}_A(R)(\sigma) = \begin{cases} \text{least } n \text{ s.t. } \mathbf{CompStep}_A(n)(R)(\sigma) = * \\ \qquad \text{if such an } n \text{ exists}, \\ \infty \qquad \text{otherwise}. \end{cases}$$

Now we can give the *operational semantics* for programs (compare [dB], Definition 5.17), with the function

$$O_A : \mathbf{ClProg} \to (\mathrm{PR.STATE}(A) \dashrightarrow \mathrm{STATE}(A)),$$

so that $O_A(R)$ is a partial function from proper states to states, defined by

$$O_A(R)(\sigma) \simeq \begin{cases} \mathbf{CompStep}_A(\mathbf{LengthComp}_A(R)(\sigma)-1)(R)(\sigma) \\ \qquad\qquad \text{if } \mathbf{LengthComp}_A(R)(\sigma) \neq \infty, \\ \uparrow \qquad\qquad \text{otherwise.} \end{cases}$$

Finally, we are in a position to define the function $\mathbf{Comp}_A$ which we considered at the start of this subsection; namely, $\mathbf{Comp}_A(R)(\sigma)$ is the unique computation sequence τ of length

$$\mathbf{lh}(\tau) = \mathbf{LengthComp}_A(R)(\sigma)$$

such that for all $n < \mathbf{lh}(\tau)$,

$$\tau(n) = \mathbf{CompStep}_A(n)(R)(\sigma).$$

Actually we have no use for $\mathbf{Comp}_A$! In the sequel (notably Section 3.6) the function $\mathbf{CompStep}_A$ will be used instead.

REMARK. This approach is an improvement over that in Zucker [1980], where the Recursion Theorem was needed to prove expressibility of the weakest precondition and strongest postcondition. In the present treatment the Recursion Theorem is not needed, because of the use of the function $\mathbf{CompStep}_A$ instead of $\mathbf{Comp}_A$.

3.1.6 Alternative definition of $\mathbf{CompStep}_A$

Notice that the definition of $\mathbf{CompStep}_A$ in the last subsection uses "course-of-value recursion" on the first argument; specifically, in the case of composition (Case 2), $\mathbf{CompStep}_A(n)$ depends on $\mathbf{CompStep}_A(n')$ for some $n' < n$. This is adequate for a first encounter with the semantics of recursion, but for our later work in this chapter (Section 3.6) it will prove unsatisfactory.

We therefore present an alternative definition of $\mathbf{CompStep}_A$ by primitive recursion, *i.e.*, in which $\mathbf{CompStep}_A(n)$ (for $n > 0$) depends only on $\mathbf{CompStep}_A(n-1)$.

First we define two functions, $\mathbf{first}_A$ and $\mathbf{rest}_A$ of type

$$\mathbf{ClProg} \to (\mathrm{PR.STATE}(A) \to \mathbf{Statemt}).$$

The idea is that for a program $<D \mid S>$ and proper state σ,

$\mathit{first}_A(D \mid S)(\sigma)$ and $\mathit{rest}_A(D \mid S)(\sigma)$ are statements S' and S'' respectively such that $< D \mid S' >$ gives the *first* step in the execution of $< D \mid S >$ in state σ, and $< D \mid S'' >$ gives the *rest* of the execution.

For the definition of first_A, it is convenient to adjoin a statement 'abort' to our programming language, with the meaning function $\mathcal{M}_A$ of Chapter 1 extended by:

$$\mathcal{M}_A(\mathsf{abort})(\sigma) = \varepsilon.$$

(The 'abort' construct, although convenient, is not essential, since it could be modelled by a statement of the form '$v := w$', where w is a new, and hence uninitialized, variable.)

Define a statement to be *atomic* if it is an assignment, skip or abort.

The definitions of $\mathit{first}_A(D \mid S)(\sigma)$ and $\mathit{rest}_A(D \mid S)(\sigma)$ proceed by induction on $\mathit{compl}(S)$.

Case 1. S is atomic.

$$\mathit{first}_A(D \mid S)(\sigma) \equiv S,$$
$$\mathit{rest}_A(D \mid S)(\sigma) \equiv \mathsf{skip}.$$

Case 2. $S \equiv S_1; S_2$ (the interesting case!).

$$\mathit{first}_A(D \mid S)(\sigma) \equiv \mathit{first}_A(D \mid S_1)(\sigma),$$
$$\mathit{rest}_A(D \mid S)(\sigma) \equiv \begin{cases} S_2 & \text{if } S_1 \text{ is atomic} \\ \mathit{rest}_A(D \mid S_1)(\sigma); S_2 & \text{otherwise} \end{cases}$$

Case 3. $S \equiv \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$.

$$\mathit{first}_A(D \mid S)(\sigma) \equiv \begin{cases} \mathsf{skip} & \text{if } \mathcal{R}_A(b)(\sigma) = \mathrm{tt} \text{ or } \mathrm{ff} \\ \mathsf{abort} & \text{if } \mathcal{R}_A(b)(\sigma) = \mathrm{u}, \end{cases}$$
$$\mathit{rest}_A(D \mid S)(\sigma) \equiv \begin{cases} S_1 & \text{if } \mathcal{R}_A(b)(\sigma) = \mathrm{tt} \\ S_2 & \text{if } \mathcal{R}_A(b)(\sigma) = \mathrm{ff} \\ \mathsf{skip} & \text{if } \mathcal{R}_A(b)(\sigma) = \mathrm{u}. \end{cases}$$

Case 4. $S \equiv P_i$ $(1 \leq i \leq m$, where $D \equiv < P_j \Leftarrow S_j >_{j=1}^{m})$.

$$\mathit{first}_A(D \mid S)(\sigma) \equiv \mathsf{skip},$$
$$\mathit{rest}_A(D \mid S)(\sigma) \equiv S_i.$$

This completes the definition of first_A and rest_A.

Note that (for $\sigma \neq \varepsilon$) $\mathit{first}_A(D \mid S)(\sigma)$ is always atomic, and so $\mathcal{M}_A(\mathit{first}_A(D \mid S)(\sigma))(\sigma)$ is well-defined, where (as before) $\mathcal{M}_A$ is the

meaning function of 1.2.11, extended by $\mathcal{M}_A(\textsf{abort})(\sigma) = \varepsilon$.

The promised definition of $\textit{CompStep}_A(n)(D \mid S)(\sigma)$ now follows simply, by induction on n. For $n=0$, we have as before

$$\textit{CompStep}_A(0)(D \mid S)(\sigma) = \sigma .$$

Otherwise, put

$$\sigma' =_{df} \mathcal{M}_A(\textit{first}_A(D \mid S)(\sigma))(\sigma).$$

Then for $n=1$,

$$\textit{CompStep}_A(1)(D \mid S)(\sigma) = \sigma'$$

and for $n>1$,

$$\textit{CompStep}_A(n)(D \mid S)(\sigma) = \begin{cases} \textit{CompStep}_A(n-1)(D \mid \textit{rest}_A(D \mid S)(\sigma))(\sigma') \\ \qquad \text{if } S \text{ is not atomic and } \sigma' \neq \varepsilon, \\ * \qquad \text{otherwise.} \end{cases}$$

The above definition of $\textit{CompStep}_A$ is equivalent to that in 3.1.5, in the following sense. Let us denote (temporarily) by $\textit{CompStep}'_A$, the function as defined in 3.1.5.

PROPOSITION. $\textit{CompStep}_A(n)(D \mid S) = \textit{CompStep}'_A(n)(D \mid S)$.

PROOF. Exercise. Use induction on $(n, \textit{compl}(S))$. □

3.1.7 Properties of O_A

The function O_A satisfies various desired properties (*cf.* 1.2.11 and 2.2.3).

THEOREM. (1) *For S a* skip *or assignment, $O_A(D \mid S) = \mathcal{M}_A(S)$, where $\mathcal{M}_A$ is the function of Chapter 1.*

(2) $$O_A(D \mid S_1; S_2)(\sigma) \simeq \begin{cases} O_A(D \mid S_2)(\sigma') & \text{if } O_A(D \mid S_1)(\sigma) \downarrow \sigma' \neq \varepsilon, \\ \varepsilon & \text{if } O_A(D \mid S_1)(\sigma) \downarrow \varepsilon, \\ \uparrow & \text{if } O_A(D \mid S_1)(\sigma) \uparrow \end{cases}$$

(3) $$O_A(D \mid \textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \textsf{ fi})(\sigma) \simeq \begin{cases} O_A(D \mid S_1)(\sigma) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{t} \\ O_A(D \mid S_2)(\sigma) & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{f} \\ \varepsilon & \text{if } \mathcal{R}_A(b)(\sigma) = \mathbb{u} \end{cases}$$

(4) $O_A(D \mid P_i) \simeq O_A(D \mid S_i)$ *for* $i = 1, \ldots, m$, *where* $D \equiv \langle P_j \Leftarrow S_j \rangle_{j=1}^{m}$.

PROOF. Directly from the definitions of $\textit{CompStep}_A$ and O_A. □

3.1.8 Denotational semantics

We will define, for each $A \in \mathbb{K}$, a function

$$\mathcal{M}_A : \mathit{CLProg} \to (\mathrm{PR.STATE}(A) \dashrightarrow \mathrm{STATE}(A)),$$

giving the denotational semantics for programs. Our approach is similar to that of Apt [1981] (in that we do not use environments, as in [dB]).

First we define a partial order on the set of partial state transformers on A.

DEFINITION 1. (*a*) ST.TRANS(A) is the set of partial state transformers on A (see 2.6.1).

(*b*) The partial order '$\sqsubseteq_A$' on ST.TRANS(A) is defined by: $\Phi_1 \sqsubseteq_A \Phi_2$ iff for all $\sigma \in \mathrm{PR.STATE}(A)$,

$$\Phi_1(\sigma)\downarrow \;\Rightarrow\; (\Phi_2(\sigma)\downarrow \text{ and } \Phi_2(\sigma) = \Phi_1(\sigma)),$$

i.e., $\Phi_1 \subseteq \Phi_2$ as sets of ordered pairs. (Don't confuse this with the ordering '$\subsetneq$' between states defined in 1.2.7 and 1.2.12.)

PROPOSITION 1. *The structure* (ST.TRANS(A), $\sqsubseteq_A$) *is a complete partially ordered set.*

In other words ([dB], p. 73),

(a) there is a $\sqsubseteq_A$-least element, namely the "totally undefined transformer" $\Phi_\perp$, where $\Phi_\perp(\sigma)\uparrow$ for all $\sigma \neq \varepsilon$, and

(b) any $\sqsubseteq_A$-increasing sequence $\Phi_0 \sqsubseteq_A \Phi_1 \sqsubseteq_A \cdots$ has a $\sqsubseteq_A$-supremum $\Phi = \bigsqcup_{n=0}^{\infty} \Phi_n$, defined by: for all $\sigma \neq \varepsilon$ and σ',

$$\Phi(\sigma)\downarrow\sigma' \iff \text{for some } n,\ \Phi_n(\sigma)\downarrow\sigma'$$

(*i.e.*, Φ is the set-theoretic union of the Φ_n's, considered as sets of ordered pairs).

REMARK. For a sequence $\Phi_0 \sqsubseteq_A \Phi_1 \sqsubseteq_A \cdots$, it does *not* follow, from the fact that each Φ_n is finitely based (see 2.6.1, Definition (2)), that $\bigsqcup_n \Phi_n$ is finitely based.

Now, given a (closed) program $R \equiv \langle D \mid S \rangle$, where $D \equiv \langle P_j \Leftarrow S_j \rangle_{j=1}^{m}$, we will define $\mathcal{M}_A(R)$ as the limit, or least upper bound (in the sense of Proposition 1) of a sequence of partial state transformers $\mathcal{M}_A^n(R)$, where, for each $n \geq 0$, $\mathcal{M}_A^n(R)$ is the approximate meaning of R given by interpreting procedures calls of depth n or more simply as diverging.

DEFINITION 2. $\mathcal{M}_A^n(R)$, for $n=0,1,\ldots$ and $R=(D\mid S)$, is defined by induction on $(n, \mathit{compl}(S))$.

Basis $(n=0)$. For S a **skip**, assignment, composition or conditional statement, $\mathcal{M}_A^0(D\mid S)$ is defined just like $\mathcal{M}_A(S)$ in Chapter 2 (see 2.2.3). And for S a procedure variable P_i $(i=1,\ldots,m)$ and $\sigma\neq\varepsilon$, we have

$$\mathcal{M}_A^0(D\mid P_i)(\sigma)\uparrow.$$

Induction step. Again, for S a **skip**, assignment, composition or conditional statement, $\mathcal{M}_A^{n+1}(D\mid S)$ is defined just like $\mathcal{M}_A(S)$ in Chapter 2; and further, for $i=1,\ldots,n$ and $\sigma\neq\varepsilon$,

$$\mathcal{M}_A^{n+1}(D\mid P_i)=\mathcal{M}_A^n(D\mid S_i)$$

(where $D\equiv <P_j \Leftarrow S_j>_{j=1}^m$).

PROPOSITION 2. *The sequence* $(\mathcal{M}_A^n(R))_n$ *is* $\sqsubseteq_A$*-increasing.*

PROOF. Show that $\mathcal{M}_A^n(D\mid S)\sqsubseteq_A \mathcal{M}_A^{n+1}(D\mid S)$, by induction on $(n,\mathit{compl}(S))$. □

This proposition justifies the following definition.

DEFINITION 3. $\mathcal{M}_A(R)=_{df}\bigsqcup_{n=0}^{\infty}\mathcal{M}_A^n(R)$.

In other words, $\mathcal{M}_A(R)(\sigma)\downarrow\sigma'$ iff for some n, $\mathcal{M}_A^n(R)(\sigma)\downarrow\sigma'$, or equivalently, iff for some m and all $n>m$, $\mathcal{M}_A^n(R)(\sigma)\downarrow\sigma'$.

3.1.9 Properties of $\mathcal{M}_A$

The function $\mathcal{M}_A$ satisfies various desired propeties.

THEOREM. *The statements in the Theorem in 3.1.7 hold, with '*$\mathcal{M}_A$*' replacing '*O_A*' throughout.*

PROOF. For (1), (2) and (3): these hold for $\mathcal{M}_A^n$ $(n=0,1,\ldots)$ by definition. Hence they hold for $\mathcal{M}_A$, by taking suprema.

As for (4), namely the *fixed-point property* of $\mathcal{M}_A$:

$$\begin{aligned}\mathcal{M}_A(D\mid P_i) &= \bigsqcup_{n=0}^{\infty}\mathcal{M}_A^n(D\mid P_i)\\ &= \bigsqcup_{n=1}^{\infty}\mathcal{M}_A^n(D\mid P_i) \qquad (\text{since } (\mathcal{M}_A^n(D\mid P_i))_n \text{ is increasing})\\ &= \bigsqcup_{k=0}^{\infty}\mathcal{M}_A^{k+1}(D\mid P_i)\end{aligned}$$

$$= \bigsqcup_{k=0}^{\infty} \mathcal{M}_A^k(D \mid S_i) \quad \text{(by definition)}$$

$$= \mathcal{M}_A(D \mid S_i). \quad \square$$

3.1.10 Equivalence of operational and denotational semantics

THEOREM. $O_A(R) = \mathcal{M}_A(R)$.

PROOF. As in the proof of [dB], Theorem 5.22, one shows:

(1) $O_A(R) \sqsubseteq_A \mathcal{M}_A(R)$, and

(2) $\mathcal{M}_A(R) \sqsubseteq_A O_A(R)$.

Now (1) is proved by induction on $\mathbf{LengthComp}_A(R)(\sigma)$, using the Theorems in 3.1.7 and 3.1.9, and (2) is proved by showing that for all n, $\mathcal{M}_A^n(D \mid S) \sqsubseteq_A O_A(D \mid S)$, by induction on $(n, \mathbf{compl}(S))$.

3.1.11 Monotonicity for statements

(Compare 1.2.13 and 2.2.8.)

THEOREM. *Suppose* $M \supseteq \mathbf{ProgVar}(R)$, $\sigma, \sigma' \neq \varepsilon$ *and* $\sigma \subsetneq \sigma'$ (rel M). *Then:*
(1) *If* $\mathcal{M}_A(R)(\sigma)\downarrow \neq \varepsilon$ *then also* $\mathcal{M}_A(R)(\sigma')\downarrow \neq \varepsilon$ *and* $\mathcal{M}_A(R)(\sigma) \subsetneq \mathcal{M}_A(R)(\sigma')$ (rel M).
(2) *If* $\mathcal{M}_A(R)(\sigma)\uparrow$ *then also* $\mathcal{M}_A(R)(\sigma')\uparrow$.

PROOF. *Either* use the operational semantics, proving a suitable monotonicity result for $\mathbf{CompStep}_A(n)(D \mid S)$, by induction on $(n, \mathbf{compl}(S))$, and then inferring the theorem for $O_A(R)$;

Or use the denotational semantics, proving the result for $\mathcal{M}_A^n(D \mid S)$, by induction on $(n, \mathbf{compl}(S))$, and then taking suprema. $\square$

COROLLARY. *Suppose* $M \supseteq \mathbf{ProgVar}(R)$, $\sigma, \sigma' \neq \varepsilon$ *and* $\sigma \simeq \sigma'$ (rel M). *Then either*
(i) $\mathcal{M}_A(R)(\sigma)\downarrow \neq \varepsilon$, $\mathcal{M}_A(R)(\sigma')\downarrow \neq \varepsilon$ *and* $\mathcal{M}_A(R)(\sigma) \simeq \mathcal{M}_A(R)(\sigma')$ (rel M), *or*
(ii) $\mathcal{M}_A(R)(\sigma)\downarrow \varepsilon$ *and* $\mathcal{M}_A(R)(\sigma')\downarrow \varepsilon$, *or*
(iii) $\mathcal{M}_A(R)(\sigma)\uparrow$ *and* $\mathcal{M}_A(R)(\sigma')\uparrow$.

3.1.12 Variables in the "left hand side" of a program; constancy of other variables

(Compare 1.2.14, 2.2.9.) The definition of $\mathbf{lhs}(R)$ augments that of $\mathbf{lhs}(S)$ in 1.2.14 with the new clauses:

$$\mathbf{lhs}(P) = \emptyset$$

$$\mathbf{lhs}(D) = \bigcup_{i=1}^{m} \mathbf{lhs}(S_i), \quad \text{where } D \equiv < P_i \Leftarrow S_i >_{i=1}^{m}$$

$$\mathbf{lhs}(D \mid S) = \mathbf{lhs}(D) \cup \mathbf{lhs}(S).$$

Again we have $\mathbf{lhs}(R) \subseteq \mathbf{ProgVar}(R)$, and

THEOREM. *Suppose* $\sigma \neq \varepsilon$ *and* $\mathcal{M}_A(R)(\sigma)\downarrow\sigma' \neq \varepsilon$. *If* V *is not in* $\mathbf{lhs}(R)$ *then* $\sigma'(V) = \sigma(V)$.

PROOF. *Either* use the operational semantics, proving the appropriate result for $\mathbf{CompStep}_A(n)(D \mid S)$, by induction on $(n, \mathbf{compl}(S))$;

Or use the denotational semantics, proving the result for $\mathcal{M}_A^n(D \mid S)$, by induction on $(n, \mathbf{compl}(S))$. □

3.1.13 Isomorphism between structures; semantics abstraction theorem

(Compare 1.2.16 and 2.2.11.) Once again, the semantics of our programming language $\mathbf{ProgLang}_{ra}(\Sigma)$ satisfies the Program Semantics Abstraction Principle.

THEOREM. *Given a* Σ*-isomorphism* $\varphi: A \to B$ *and any* $R \in \mathbf{ClProg}(\Sigma)$ *(and with the notation as in 1.2.16), the following diagram commutes:*

$$\begin{array}{ccc} \text{PR.STATE}(A) & \xrightarrow{\mathcal{M}_A(R)} & \text{STATE}(A) \\ \Big\downarrow \hat{\varphi} & & \Big\downarrow \hat{\varphi}_\varepsilon \\ \text{PR.STATE}(B) & \xrightarrow{\mathcal{M}_B(R)} & \text{STATE}(B) \end{array}$$

PROOF. Use either the operational or the denotational semantics, as in the proofs of the theorems in 3.1.11 and 3.1.12. □

3.2 ASSERTIONS

3.2.1 Structures A^* of signature Σ^*

The assertions for recursive programs are again given in $\boldsymbol{Lang}_1(\Sigma^*)$, the first order language over Σ^*, as defined in Chapter 2.

However there is a new factor which was not present in Chapter 2: we will need to use the structures A^* of signature Σ^* as *data structures*, and consider 'while' program computations over them, for reasons to be explained below (Section 3.5).

Now, in order to ensure that certain basic operations over A^* are computable, we must enrich the signature Σ^* of Chapter 2 with some new operations. However the resulting signature, which we still call Σ^* for convenience, is an *inessential extension* of the old one, in a sense to be made precise below (3.2.3).

We now describe the new operations in Σ^*. Recall that, for $A \in \mathbb{K}$, A^* was defined (in 2.3.1) as

$$A^* = ((A_i^{\mu})_{i \in Sort}, (A_i^*)_{i \in Sort}, (F_j^{A,\mu})_{1 \le j \le s}, (\mathbb{u}_i)_{i \in Sort}, (\mathsf{Ap}_i^A)_{i \in Sort})$$

with signature Σ^*.

We need the following three new constant and operation symbols, for each sort i:

(1) the null constant Null_i, with interpretation

$$\mathsf{Null}_i^A = \lambda n \cdot \mathbb{u}_i \in A_i^*,$$

(2) the adjunction operator Adjoin_i, with interpretation

$$\mathsf{Adjoin}_i^A : A_i^* \times \mathbb{N}^{\mu} \times A_i^{\mu} \to A_i^*,$$

which adjoins a new argument and value to a function in A_i^*, as follows: for $\xi \in A_i^*$, $n \in \mathbb{N}$ and $x \in A_i^{\mu}$, $\mathsf{Adjoin}_i^A(\xi, n, x)$ is the function in A_i^* such that for all $k \in \mathbb{N}$,

$$\mathsf{Adjoin}_i^A(\xi, n, x)(k) = \begin{cases} \xi(k) & \text{if } k \neq n \\ x & \text{if } k = n, \end{cases}$$

and $\mathsf{Adjoin}_i^A(\xi, \mathbb{u}_{\mathbb{N}}, x)$ is the trivial function Null_i^A, and finally

(3) the unary relation Unspec_i, with interpretation

$$\mathsf{Unspec}_i^A : A_i^{\mu} \to \mathbb{B}^{\mu}$$

where for all $x \in A_i^{\mu}$

$$\mathsf{Unspec}_i^A(x) = \begin{cases} \mathbb{t} & \text{if } x = \mathbb{u}_i \\ \mathbb{f} & \text{otherwise.} \end{cases}$$

(This is in addition to the individual constant unspec_i.)

Then A^* is the structure

$$A^* = ((A_i^{\mathbb{u}})_{i \in Sort}, (A_i^*)_{i \in Sort}, (F_j^{A,\mathbb{u}})_{1 \le j \le s}, (\mathbb{u}_i)_{i \in Sort}, (\mathsf{Ap}_i^A)_{i \in Sort},$$
$$(\mathsf{Null}_i^A)_{i \in Sort}, (\mathsf{Adjoin}_i^A)_{i \in Sort}, (\mathsf{Unspec}_i^A)_{i \in Sort})$$

with signature Σ^*.

REMARKS. (1) As stated above, we will later use the structures A^* as data structures, and consider 'while' computations over them (Section 3.5). Then we will need these three new operations Null_i, Adjoin_i and Unspec_i.

We did not need these operations for the signature Σ^* in Chapter 2, since there we did not have to consider computations over A^*, and Σ^* was kept as simple as possible.

We will, however, keep the same notation (Σ^* and A^*) for the enriched signature and structures as for those of Chapter 2, since this enrichment of Σ^* is inessential, as we will see (3.2.3).

(2) We could have introduced, for each sort i, instead of Unspec_i, the identity relation

$$\mathsf{Ident}_i^A : A_i^{\mathbb{u}} \times A_i^{\mathbb{u}} \to \mathbb{B}^{\mathbb{u}}$$

where, for all $x, y \in A_i^{\mathbb{u}}$,

$$\mathsf{Ident}_i^A(x, y) = \begin{cases} \mathbb{t} & \text{if } x = y \\ \mathbb{f} & \text{otherwise.} \end{cases}$$

(Note that this differs from the extended equality relation $\mathsf{eq}_i^{A,\mathbb{u}}$ for $i = \mathsf{N}$ or B, since $\mathsf{eq}_i^{A,\mathbb{u}}(x, y) = \mathbb{u}_i$ whenever $x = \mathbb{u}_i$ or $y = \mathbb{u}_i$, whether $x = y$ or not; see 1.1.5.)

Then we could define

$$\mathsf{Unspec}_i(x) \equiv_{df} \mathsf{Ident}_i(x, \mathsf{unspec}_i).$$

But the assumption of a computable identity relation for any (arbitrary) domain seems unwarranted. (Why should we assume that we could effectively test for identity between, for example, two elements of A_i^*?) However, the assumption that one can distinguish any object in a given domain at least from the specially defined "unspecified object" seems reasonable.

(3) Because we will use the structures A^* as data types for programs, we want Σ^* to be a signature in the same way that Σ is (in the sense of 1.1.1). This implies that Σ^* must include types for $\mathbb{N}$ and $\mathbb{B}$. Now this condition is not precisely satisfied, since the structures A^* contain domains $\mathbb{N}^u$ and $\mathbb{B}^u$ instead of $\mathbb{N}$ and $\mathbb{B}$. However this does not really matter; $\mathbb{N}^u$ and $\mathbb{B}^u$ can function quite well in place of $\mathbb{N}$ and $\mathbb{B}$, since we have the predicate Unspec_i to distinguish effectively between $\mathbb{u}_i$ and "genuine" elements of A_i (for $i = \mathsf{N}$ or B).

3.2.2 The assertion language

The assertion language $\boldsymbol{AssLang}_{ra} = \boldsymbol{AssLang}_{ra}(\Sigma)$ is $\boldsymbol{Lang}_1(\Sigma^*)$, the first order language over Σ^* (as defined in the last subsection). This is an *extension* of the assertion language of Chapter 2. There the only assertion terms of type i^* were *variables* ξ^i (see 2.3.3). Now, because of the new constants, there are, for every sort i, more general terms of type i^*:

$$s^{i*} ::= \xi^i \mid \mathsf{Null}_i \mid \mathsf{Adjoin}_i(s^{i*}, s^{\mathsf{N}}, s^i).$$

There are also new formation rules for assertion terms of type i:

$$s^i ::= \cdots \mid \mathsf{Ap}_i(s^{i*}, s^{\mathsf{N}})$$

and, further, for assertion booleans (*i.e.* assertion terms of type B):

$$s^{\mathsf{B}} ::= \cdots \mid \mathsf{Unspec}_i(s^i).$$

The class $\boldsymbol{Assn}^* = \boldsymbol{Assn}^*(\Sigma)$ of assertions $p, q, r, \ldots$ is then defined as in 2.3.3, relative to these assertion terms.

As before, we will use the notation

$$\boldsymbol{Var}(E) \equiv \boldsymbol{AssVar}(E) \cup \boldsymbol{ProgVar}(E)$$

for any syntactic expression E, where $\boldsymbol{AssVar}(E)$ is the set of *free* assertion variables in E.

The *semantics* of the assertion language, specifically the definition of evaluation functions $\mathcal{S}_A$ and $\mathcal{J}_A$, extends that of Chapter 2 (see 2.3.5) in the obvious way, according to the stated interpretations of the new symbols Null_i, Adjoin_i and Unspec_i.

The notions of *interpretation*, *satisfaction* ($I \models p$) and $\mathbb{K}$-*validity* ($\mathbb{K} \models p$) are again defined as in 1.3.11.

3.2.3 An inessential extension

The extension to the assertion language of Chapter 2 formed by adding the new symbols Null_i, Adjoin_i and Unspec_i is *inessential*, in the sense of the following theorem.

THEOREM. *For any assertion p, in the extended language described above, we can effectively find an assertion $\tilde{p}$ not containing any of the symbols Null_i, Adjoin_i and Unspec_i, such that*

$$\mathbb{K} \models p \leftrightarrow \tilde{p} .$$

PROOF. We describe the construction of $\tilde{p}$ from p. First, define a *big term* to be an assertion term of type i^* (for some sort i) which is not a variable. Note that all big terms are of the form *either* Null_i *or* $\mathsf{Adjoin}_i(s^{i*}, s^{\mathsf{N}}, s^i)$. We proceed in two stages.

Stage 1: the elimination from p of all occurrences of the symbols Null_i and Adjoin_i. This amounts to the elimination of all big terms from p. This is done as follows. Let s^{i*} be a *maximal* occurrence of a big term in p (*i.e.* not occurring within another big term). Then s^{i*} occurs in the context $\mathsf{Ap}_i(s^{i*}, s^{\mathsf{N}})$. There are two cases.

Case 1. $s^{i*} \equiv \mathsf{Null}_i$. Then replace $\mathsf{Ap}_i(s^{i*}, s^{\mathsf{N}})$ by unspec_i.

Case 2. $s^{i*} \equiv \mathsf{Adjoin}_i(s_0^{i*}, s_0^{\mathsf{N}}, s_0^i)$. Then replace $\mathsf{Ap}_i(s^{i*}, s^{\mathsf{N}})$ by the term

$$\mathsf{if}\ \mathsf{eq}_{\mathsf{N}}(s^{\mathsf{N}}, s_0^{\mathsf{N}})\ \mathsf{then}\ s_0^i\ \mathsf{else}\ \mathsf{Ap}_i(s_0^{i*}, s^{\mathsf{N}})\ \mathsf{fi}.$$

This process is repeated until there are no occurrences of Null_i or Adjoin_i left in p.

Stage 2: the elimination of all occurrences of Unspec_i. Consider a maximal assertion boolean in p of the form $s^{\mathsf{B}} \equiv \mathsf{Unspec}_i(s_0^i)$. Let p_0 be an atomic assertion in p containing s^{B}. Now replace p_0 by the assertion

$$\exists x^{\mathsf{B}}[((s_0^i = \mathsf{unspec}_i \wedge x^{\mathsf{B}} = \mathsf{true}) \vee (s_0^i \neq \mathsf{unspec}_i \wedge x^{\mathsf{B}} = \mathsf{false}))$$
$$\wedge\ p_0 < x^{\mathsf{B}}/s^{\mathsf{B}} >],$$

where $x^{\mathsf{B}} \notin \mathit{Var}(p_0)$, and $p_0 < x^{\mathsf{B}}/s^{\mathsf{B}} >$ is the result of substituting x^{B} for s^{B} in p_0 (which is well defined, since p_0 is atomic).

This process is repeated until there are no occurrences of Unspec_i left in p, resulting in the assertion $\tilde{p}$. □

3.2.4 Invariance for assertions

THEOREM. *If* $\mathbf{Var}(s,p) \cap \mathbf{lhs}(R) = \emptyset$, $\sigma \neq \varepsilon$ *and* $\mathfrak{M}_A(R)(\sigma)\downarrow\sigma' \neq \varepsilon$, *then*
(*a*) $\mathcal{S}_A(s)(\rho,\sigma') = \mathcal{S}_A(s)(\rho,\sigma)$, *and*
(*b*) $\mathcal{T}_A(p)(\rho,\sigma') = \mathcal{T}_A(p)(\rho,\sigma)$.

PROOF. (*a*) Induction on $\mathbf{compl}(s)$. For the basis, use the theorem in 3.1.12. (*b*) Induction on $\mathbf{compl}(p)$. For the basis, use part (*a*). □

This theorem will be used in the proof of the validity of the invariance rule (3.4.2).

3.3 CORRECTNESS FORMULAE

3.3.1 Syntax

We define three syntactic classes.

(1) The class $\mathbf{Form}_{ra} = \mathbf{Form}_{ra}(\Sigma)$ of *correctness formulae* for recursive programs (relative to Σ), denoted $f, \ldots$, is defined as before (1.4.1, 2.4.1) by

$$f ::= \{p\}S\{q\} \mid p\,.$$

(2) The class of *declared correctness formulae* consists of expressions of the form

$$< D \mid f >$$

which are *closed*, in the sense that if $D \equiv < P_i \Leftarrow S_i >_{i=1}^{m}$, then all the procedure variables occurring in f or in any of the S_i are among $P_1, \ldots, P_m$ (*cf.* Convention 3.1.4).

(3) The class of *sequents of correctness formulae*, denoted $g, \ldots$, is defined by

$$g ::= < D \mid \vec{f} \rightarrow f >$$

where $\vec{f} \equiv f_1, \ldots, f_k$ $(k \geq 0)$ is a vector of correctness formulae. Again g is assumed to be *closed*, in the sense that if $D \equiv < P_i \Leftarrow S_i >_{i=1}^{m}$, then all the procedure variables occurring in $\vec{f}$, f or any of the S_i are among $P_1, \ldots, P_m$.

3.3.2 Semantics

The notion of *interpretation in* $\mathbb{K}^*$, $I=(A^*,\rho,\sigma)\in \mathrm{INTERP}^*(\mathbb{K})$, is defined as in Chapter 2 (2.3.5).

DEFINITION 1 (*n-satisfaction of declared correctness formulae*).
We define, for $n=0,1,2,\ldots$, the notion $I\models^n <D\,|\,f>$.

Case 1. $f\equiv\{p\}S\{q\}$. Then $A^*,\rho,\sigma\models^n <D\,|\,f>$ iff for all σ':

$$(A^*,\rho,\sigma\models p \text{ and } \mathcal{M}^n_A(D\,|\,S)(\sigma)\downarrow\sigma') \Rightarrow (\sigma'\neq\varepsilon \text{ and } A^*,\rho,\sigma'\models q).$$

(Recall Definition 2 in 3.1.8.)

Case 2. $f\equiv p$. Then $I\models^n <D\,|\,f>$ iff $I\models f$ according to the definition in 3.2.2.

Note that the only difference from the definition of $I\models f$ in Chapter 2 (2.4.2, Definition (1)) lies in the presence of the superscript 'n' in $\mathcal{M}^n_A$ (and, of course, the presence of the declaration D).

DEFINITION 2 ($\mathbb{K}$, *n-validity of declared correctness formulae*).
For $n=0,1,2,\ldots$, $\mathbb{K}\models^n <D\,|\,f>$ iff

$$I\models^n <D\,|\,f> \text{ for all } I\in \mathrm{INTERP}^*(K).$$

DEFINITION 3 ($\mathbb{K}$, *n-validity of sequents*). For $n=0,1,2,\ldots$,
$\mathbb{K}\models^n <D\,|\,f_1,\ldots,f_k\rightarrowtriangle f>$ iff

$$(\mathbb{K}\models^n <D\,|\,f_i> \text{ for } i=1,\ldots,k) \Rightarrow \mathbb{K}\models^n <D\,|\,f>.$$

DEFINITION 4 ($\mathbb{K}$-*validity of sequents*). $\mathbb{K}\models g$ iff

$$\mathbb{K}\models^n g \quad \text{for all } n.$$

NOTATION. $\mathbb{K}\models^n <D\,|\,\vec{f}>$ denotes $\mathbb{K}\models^n <D\,|\,f_i>$ for $i=1,\ldots,k$, where $\vec{f}\equiv f_1,\ldots,f_k$ (and similarly with '$\mathbb{K}\models$' in place of '$\mathbb{K}\models^n$').

3.3.3 Remarks on the validity of sequents without antecedents

Consider a sequent $g\equiv <D\,|\,\vec{f}\rightarrowtriangle f>$ where $\vec{f}$ is empty. We will write g as $<D\,|\,f>$, *i.e.*, like a declared correctness formula. Now, by definition of $\mathbb{K}$-validity, we have that g is $\mathbb{K}$-valid if and only if

$$\text{for all } n,\ \mathbb{K}\models^n <D\,|\,f>,$$

and, assuming $f\equiv\{p\}S\{q\}$, this is equivalent to:

$$\text{for all } n,\ I\equiv(A^*,\rho,\sigma) \text{ and } \sigma'\neq\varepsilon,$$
$$(I\models p \text{ and } \mathcal{M}^n_A(D\,|\,S)(\sigma)\downarrow\sigma') \Rightarrow A^*,\rho,\sigma'\models q,$$

which (by the remark in 3.1.8 after Definition 3) is equivalent to:

$$\text{for all } I \equiv (A^*, \rho, \sigma) \text{ and } \sigma' \neq \varepsilon,$$
$$(I \models p \text{ and } \mathcal{M}_A(D \mid S)(\sigma)\downarrow\sigma') \Rightarrow A^*, \rho, \sigma' \models q,$$

which resembles the definition of $\mathbb{K}$-validity of correctness formulae in Chapter 2 (2.4.2).

Note however that in the general case, with $g \equiv \langle D \mid f_1, \ldots, f_k \rightarrowtriangle f \rangle$ $(k > 0)$, $\mathbb{K}$-validity of g is *not* equivalent to:

$$(\mathbb{K} \models \langle D \mid f_i \rangle \text{ for } i = 1, \ldots, k) \Rightarrow \mathbb{K} \models \langle D \mid f \rangle. \qquad (1)$$

In fact, the recursion rule (Section 3.4) is not valid for the notion of validity given by (1).

3.3.4 A lemma on n-Satisfaction

LEMMA. $I \models^{n+1} \langle D \mid f \rangle \Rightarrow I \models^n \langle D \mid f \rangle$ $(n = 0, 1, , \ldots)$.

PROOF. Use the fact that $\mathcal{M}_A^n(D \mid S)(\sigma)\downarrow\sigma'$ implies $\mathcal{M}_A^{n+1}(D \mid S)(\sigma)\downarrow\sigma'$ (by 3.1.8, Proposition 2). □

This will be used to prove the soundness of the recursion rule (3.4.2).

3.3.5 Substitution in correctness formulae; Substitution theorems

We develop the notion of substitution of one assertion variable for another, considered in 1.3.16.

We first extend this notion to correctness formulae by defining:

$$\{p\}S\{q\}\langle y/x\rangle \equiv_{df} \{p\langle y/x\rangle\}S\{q\langle y/x\rangle\}$$

where x and y are assertion variables of the same type (i or i^* for any sort i).

Let us use the following notation, for given (fixed) x and y. For any syntactic expression E, let $\tilde{E} \equiv_{df} E\langle y/x\rangle$, and for any valuation ρ and interpretation $I = (A^*, \rho, \sigma)$, let $\tilde{\rho} =_{df} \rho\{\rho(y)/x\}$ and $\tilde{I} =_{df} (A^*, \tilde{\rho}, \sigma)$.

THEOREM (Substitution for assertion variables in correctness formulae).

(*a*) $I \models \tilde{p}$ *iff* $\tilde{I} \models p$.

(*b*) $I \models^n \langle D \mid \tilde{f} \rangle$ *iff* $\tilde{I} \models^n \langle D \mid f \rangle$.

PROOF. Part (*a*) follows easily from the corresponding theorem in 1.3.16. Part (*b*) then follow from part (*a*). □

This will be used to prove the soundness of the substitution rule (3.4.2).

3.4 A PROOF SYSTEM; SOUNDNESS

3.4.1 The proof system

We define a proof system $\boldsymbol{ProofSys}_{ra} = \boldsymbol{ProofSys}_{ra}(\mathbb{K})$ for deriving $\mathbb{K}$-valid sequents. The nodes of the proof trees consist now of sequents.

The proof rules can again be divided into three groups (see $\boldsymbol{ProofSys}_{sa}$, 1.5.1.).

(A) *Rules for the programming language constructs.*

First, there are rules corresponding to the (A)-rules of $\boldsymbol{ProofSys}_{sa}$ ('skip', *assignment*, *composition* and *conditional*) which are formed as follows. If

$$\frac{f_1, \ldots, f_n}{f} \qquad (0 \leq n \leq 3)$$

is one of the rules of $\boldsymbol{ProofSys}_{sa}$, then the corresponding rule for the present system is:

$$\frac{< D \mid \vec{f} \rightarrow f_1 >, \ldots, < D \mid \vec{f} \rightarrow f_n >}{< D \mid \vec{f} \rightarrow f >}$$

for any vector $\vec{f}$ of correctness formulae.

Thus, for example, the *composition* rule now has the form

$$\frac{< D \mid \vec{f} \rightarrow \{p\} S_1 \{r\} >, \quad < D \mid \vec{f} \rightarrow \{r\} S_2 \{q\} >}{< D \mid \vec{f} \rightarrow \{p\} S_1; S_2 \{q\} >}.$$

Further, we have:

(A.5) *Recursion*:

Suppose $D \equiv < P_i \Leftarrow S_i >_{i=1}^{m}$. Then we have the following m rules, all with the same m premisses:

$$\frac{< D \mid \vec{f}, \{p_1\} P_1 \{q_1\}, \ldots, \{p_m\} P_m \{q_m\} \rightarrow \{p_j\} S_j \{q_j\} > \quad \text{for } j = 1, \ldots, m}{< D \mid \vec{f} \rightarrow \{p_i\} P_i \{q_i\} >}$$

for $i = 1, \ldots, n$. This is a version of *Scott's induction rule* (see Chapter 5 in [dB] and the references R.3.4 in [dB], p. 472).

This can be considered as a rule for the programming language construct *procedure variable*, and is hence put in group (A).

(B) *The* $\mathbb{K}$*-oracle rule*:

This now has the form

$$< D \mid \vec{f} \rightarrow\!\triangleright\ p >$$

for any $\mathbb{K}$-valid assertion p and any vector $\vec{f}$ of correctness formulae.

(C) *Logical rules*.

In addition to the *consequence* rule, modified exactly like the rules in group (A), we have:

(C.2) *Invariance*:

$$\frac{< D \mid \vec{f} \rightarrow\!\triangleright \{p\}S\{q\} >}{< D \mid \vec{f} \rightarrow\!\triangleright \{p \wedge r\}S\{q \wedge r\} >}$$

where $\boldsymbol{Var}(r) \cap \boldsymbol{lhs}(D,S) = \emptyset$ (or, equivalently, $\boldsymbol{ProgVar}(r) \cap \boldsymbol{lhs}(D,S) = \emptyset$).

Notice that the following form of the rule

$$< D \mid \vec{f} \rightarrow\!\triangleright \{p\}S\{p\} >$$

(where $\boldsymbol{Var}(p) \cap \boldsymbol{lhs}(D,S) = \emptyset$) is not, in general, valid! (Hint: consider the case that execution of S produces the error state.)

(C.3) *Substitution*:

$$\frac{< D \mid \vec{f} \rightarrow\!\triangleright f >}{< D \mid \vec{f} \rightarrow\!\triangleright f\langle y/x \rangle >}$$

where x and y are assertion variables of the same type (i or i^* for any sort i), and $f\langle y/x \rangle$ denotes the result of substituting y for all free occurrences of x in f (as in 3.3.5).

Note that we do not need to state any restrictions on the variables, as in the corresponding rule in Apt [1981], § 3.4 ("Substitution Rule I") or [dB], §5.5 ("Substitution, II"); such restrictions are automatically satisfied, because of our distinction between program and assertion variables.

(C.4) *Existential quantification*:

$$\frac{< D \mid \vec{f} \rightarrow\!\triangleright \{p\}S\{q\} >}{< D \mid \vec{f} \rightarrow\!\triangleright \{\exists x\,[p]\}S\{q\} >}$$

where x is an assertion variable of any type (i or i^*) such that $x \notin \boldsymbol{Var}(q)$.

This rule is used in place of the other substitution rule in Apt [1981], §3.4 ("Substitution Rule II") or [dB], §5.5 ("Substitution, I") in the completeness proof (Section 3.7).

There is an obvious "companion rule" for universal quantification:

$$\frac{< D \mid \vec{f} \to \{p\}S\{q\} >}{< D \mid \vec{f} \to \{p\}S\{\forall x[q]\} >}$$

where $x \notin \mathit{Var}(p)$; but we do not need this rule in the completeness proof.

In fact, as far as the completeness proof is concerned, we only need, in the last three rules (C.2)—(C.4), the special case that S is a procedure variable.

Finally, there are two "structural rules" for sequents:

(C.5) *Selection*:

$$< D \mid f_1, \ldots, f_k \to f_i > \qquad (1 \leq i \leq k)$$

(C.6) *Cut*:

$$\frac{< D \mid \vec{f} \to f_0 >, \quad < D \mid f_0, \vec{f}' \to f_1 >}{< D \mid \vec{f}, \vec{f}' \to f_1 >}.$$

3.4.2 Soundness

THEOREM. *The system* $\mathbf{ProofSys}_{ra}$ *is sound relative to* $\mathbb{K}$, *i.e. for any sequent* g,

$$\mathbb{K} \vdash g \;\Rightarrow\; \mathbb{K} \models g\,.$$

PROOF. This is proved, as before, by showing that each inference rule is $\mathbb{K}$-valid.

For rules (A.1)—(A.4), (B) and (C.1), one shows that for all n, $\mathbb{K},n$-validity of the premisses implies $\mathbb{K},n$-validity of the conclusion (as with the corresponding rules in Chapter 1, taking possible non-termination into account: see 1.5.2 and 2.5.2).

We consider the remaining rules. First, the recursion rule (A.5). Let $D \equiv < P_i \Leftarrow S_i >_{i=1}^{m}$, and assume

$$\mathbb{K} \models < D \mid \vec{f}, \{p_1\}P_1\{q_1\}, \ldots, \{p_m\}P_m\{q_m\} \to \{p_j\}S_j\{q_j\} > \qquad (1)$$

for $j = 1, \ldots, m$. We must prove that for all n,

$$\mathbb{K} \models^n < D \mid \vec{f} \to \{p_i\}P_i\{q_i\} > \qquad \text{for } i = 1, \ldots, m. \quad (2)$$

This is proved by induction on n. For $n=0$, note that (2) holds by definition (3.3.2, 3.1.8). Assume (as an induction hypothesis) that (2) holds for n, and assume that

$$\mathbb{K} \models^{n+1} < D \mid \vec{f} >. \qquad (3)$$

We must show that $\mathbb{K} \models^{n+1} < D \mid \{p_i\} P_i \{q_i\} >$ for $i = 1, \ldots, m$. (4)

By (3) and Lemma 3.3.4,

$$\mathbb{K} \models^n < D \mid \vec{f} >, \tag{5}$$

and so, by the induction hypothesis,

$$\mathbb{K} \models^n < D \mid \{p_i\} P_i \{q_i\} > \quad \text{for } i = 1, \ldots, m. \tag{6}$$

By (1),

$$\mathbb{K} \models^n < D \mid \vec{f}, \{p_1\} P_1 \{q_1\}, \ldots, \{p_m\} P_m \{q_m\} \rightarrow \{p_i\} S_i \{q_i\} >$$

for $i = 1, \ldots, m$. So by (5) and (6),

$$\mathbb{K} \models^n < D \mid \{p_i\} S_i \{q_i\} > \quad \text{for } i = 1, \ldots, m.$$

Hence (4) follows (since $\mathcal{M}_A^{n+1}(D \mid P_i) = \mathcal{M}_A^n(D \mid S_i)$ for all $A \in \mathbb{K}$).

The validity of the invariance rule (C.2) follows from the theorem in 3.2.4.

As for the substitution rule (C.3), its validity follows easily from the theorem in 3.3.5.

Next, the existential quantification rule (C.4). Assume

$$\mathbb{K} \models < D \mid \vec{f} \rightarrow \{p\} S \{q\} >. \tag{7}$$

We must show that for all n

$$\mathbb{K} \models^n < D \mid \vec{f} \rightarrow \{\exists x\, [p]\} S \{q\} >$$

where $x \notin \boldsymbol{Var}(q)$. So assume

$$\mathbb{K} \models^n < D \mid \vec{f} > \tag{8}$$

and take any interpretation (A^*, ρ, σ) such that (dropping the 'A^*')

$$\rho, \sigma \models^n \exists x\, [p] \tag{9}$$

and

$$\mathcal{M}_A^n(D \mid S)(\sigma) \downarrow \sigma' \tag{10}$$

for some σ'. We must show that $\sigma' \neq \varepsilon$ and

$$\rho, \sigma' \models^n q. \tag{11}$$

By (9), there exists $\boldsymbol{x}$ (in the domain of the same type as x) such that

$$\rho\{\boldsymbol{x}/x\}, \sigma' \models^n p. \tag{12}$$

By (7), (8), (12) and (10), $\sigma' \neq \varepsilon$ and

$$\rho\{\boldsymbol{x}/x\}, \sigma' \models^n q.$$

Hence (11) follows from Theorem 2 of 1.3.12 (which also holds in the present language), since $x \notin \mathit{Var}(q)$.

Finally, the validity of the two structural rules, selection (C.5) and cut (C.6), is easily proved. □

3.5 A LOOK AHEAD

3.5.1 The main problem, and an indication of its solution

Let us pause a moment and see what lies ahead. When we try to adapt the work of Chapter 2 (for 'while'-array programs) to the present setting, we encounter the following problem.

The main step in proving the expressibility of the weakest precondition and strongest postcondition in Chapter 2 was the construction (2.6.11, Definition 2) of the "computation predicate" $\mathsf{Compu}_S(X,Y,X')$ in the assertion language $\mathit{Lang}_1(\Sigma^*)$, which expresses: "$Y$ represents a finite computation sequence generated by S, starting in the state represented by X, and terminating in the state represented by X'."

This assertion was constructed by a straightforward induction on $\mathit{compl}(S)$.

However, if we try something similar now, we run into the same type of problem we had in defining the function Comp_A earlier in this chapter (3.1.5), specifically in the case that S is a procedure variable, where the predicate would be defined in terms of its meaning for a "more complicated" instance of S, thus (apparently) violating the inductive procedure. We therefore proceed as follows.

We will want to show that the partial functions $\mathfrak{M}_A(R)$ are *first-order expressible over* $\mathbb{K}^*$ (or: *over* A^*, uniformly for $A \in \mathbb{K}$); in other words, for every program R there is an assertion $\mathsf{io}_R(X,X')$ in $\mathit{Lang}_1(\Sigma^*)$, which expresses the input-output relation for R, in the following sense: for all A, ρ, σ and σ', if X and X' represent σ and σ' respectively relative to ρ (see 2.6.6), then

$$A^*, \rho \vDash \mathsf{io}_R(X,X') \iff \sigma \neq \varepsilon \text{ and } \mathfrak{M}_A(R)(\sigma)\downarrow\sigma'.$$

For then we can define (*cf.* 2.6.11, Definitions 4 and 5)

$$\mathsf{wp}_{ra}[R,p] \equiv_{df} \forall X, X'[\mathsf{Val}(X,\vec{v},\vec{a}) \wedge \mathsf{io}_R(X,X') \supset \supset \mathsf{Proper}(X') \wedge p[\![X']\!]]$$

and $$\mathsf{sp}_{ra}[p,R] \equiv_{df} \exists X', X[p[\![X]\!] \wedge \mathsf{io}_R(X',X) \wedge \mathsf{Val}(X,\vec{v},\vec{a})].$$

These satisfy (analogues of) 2.6.11, Theorems 4 and 5, and their Corollaries.

In order to prove the expressibility of $\mathfrak{M}_A(R)$ over $\mathbb{K}^*$, we proceed with the following plan. (Notice that $\mathfrak{M}_A(R)$ can be viewed as a partial function over A^* by the representation of states as vectors, as in 2.6.5.)

Step 1. Show that all partial functions which are *computable over* $\mathbb{K}^*$ are *first-order expressible over* $\mathbb{K}^*$.

Step 2. Show that the partial function representing $\mathfrak{M}_A(R)$ is *computable over* $\mathbb{K}^*$.

The main thing to explain now is the notion: *computability over* $\mathbb{K}^*$ (or: *over* A^*, uniformly for $A \in \mathbb{K}$). This necessitates an investigation of computability over abstract structures, or rather, over an arbitrary class $\mathbb{K}$ of such structures. Chapter 4 is devoted to this topic, and there two characterizations of "$\mathbb{K}$-computability" are given:

(1) *Inductive computability.* A system of *induction schemes* over the signature Σ is given, such that each scheme α defines, for each $A \in \mathbb{K}$, a partial function $\{\alpha\}^A$ over A. These schemes generalize the schemes for general recursive functions over $\mathbf{N}$ (Kleene [1952], §55).

(2) **'while'** *program computability.* Assuming certain conventions about input and output variables, a **'while'** program (in the language of Σ, without arrays) defines a partial function over each $A \in \mathbb{K}$.

In Chapter 4 the equivalence of these two notions of computability is proved (Sections 4.3 and 4.4).

Now (applying this equivalence result to the class of interest, $\mathbb{K}^*$) let us return to steps 1 and 2 above.

Step 1, the $\mathbb{K}^*$-expressibility of $\mathbb{K}^*$-computable functions, is simple, using the second characterization of computability above (**'while'** program computability) and the computation predicate of Chapter 2, and we carry it out below (3.5.2/3).

Step 2 will then be proved in the next section (3.6) using the *first* characterization; that is, we will show that for each program R there is an induction scheme which computes the function representing $\mathfrak{M}_A(R)$ over $\mathbb{K}^*$.

All that remains then, to complete the argument, is to show that

"inductive computability $\Rightarrow$ **'while'** program computability"

taking both sides over $\mathbb{K}^*$. This implication (in fact the bi-implication) will be proved for arbitrary classes $\mathbb{K}$ in Chapter 4.

REMARK. For a 'while' program S over $\mathbb{K}$, the input-output predicate $\mathsf{io}_S(X,X')$ can be defined very simply, as $\exists Y\,[\mathsf{Compu}_S(X,Y,X')]$ (see 2.6.11, Definition 2). In the present context, for a recursive program R over $\mathbb{K}$, the situation is more intricate. The predicate $\mathsf{io}_R(X,X')$ will be defined, as we will see, from the predicate Compu_S for some suitable 'while' program S (without arrays) over $\mathbb{K}^*$. In effect, R will be translated as S, with arrays over $A\in\mathbb{K}$ interpreted as elements of A^*.

3.5.2 Expressibility of 'while' computable functions over $\mathbb{K}$

We turn to Step 1 above, namely showing that all 'while' computable functions over $\mathbb{K}^*$ are expressible over $\mathbb{K}^*$.

In this section we actually prove:

THEOREM. *All* 'while' *computable functions over* $\mathbb{K}$ *are expressible over* $\mathbb{K}^*$.

From this, it immediately follows that all 'while' computable functions over $\mathbb{K}^*$ are expressible over $\mathbb{K}^{**}$ (the result of applying the '*' operation again to $\mathbb{K}^*$). The argument for Step 1 is completed by showing how to code $\mathbb{K}^{**}$ in $\mathbb{K}^*$, and this is done in the next subsection (3.5.3).

We have not yet given precise definitions of 'while' computability, or of expressibility over $\mathbb{K}$ or $\mathbb{K}^*$ (as will be done in Chapter 4), but these notions will become clear from the discussion in the proof below.

PROOF OF THEOREM. Let S be a statement in the (arrayless) 'while' programming language $\mathbf{ProgLang}_w(\Sigma)$, let $\vec{v}\equiv v_1,\ldots,v_n$ be an n-tuple of program variables of sorts $\vec{k}=k_1,\ldots,k_n$ respectively, and let w be a program variable of sort l. The triple $[S,\vec{v},w]$ is called an *i/o program*, and is said to compute a family of partial functions

$$\psi^A:\ A[\vec{k}]\dashrightarrow A_l$$

for $A\in\mathbb{K}$ (where $A[\vec{k}]=_{df}A_{k_1}\times\cdots\times A_{k_n}$), iff for all $A\in\mathbb{K}$, all $\vec{x}=x_1,\ldots,x_n\in A[\vec{k}]$ and all $\sigma\in\mathrm{PR.STATE}(A)$ satisfying $\sigma(v_i)=x_i$ for $i=1,\ldots,n$,

$$\psi^A(\vec{x})\downarrow y \quad\text{iff}\quad \mathcal{M}_A(S)(\sigma)\downarrow\sigma'\neq\varepsilon,\ \text{where } \sigma'(w)=y\neq\mathbb{u},$$

and

$$\psi^A(\vec{x})\uparrow \quad\text{iff}\quad \mathcal{M}_A(S)(\sigma)\uparrow.$$

(See 4.3.1.) Thus if $[S,\vec{v},w]$ is an i/o program, then for all $A\in\mathbb{K}$ and all σ such that $\sigma(v_i)\neq\mathbb{u}$ for $i=1,\ldots,n$, $\mathcal{M}_A(S)(\sigma)$ never aborts.

Note also that the functions ψ^A are uniquely determined from $[S,\vec{v},w]$, by monotonicity (2.2.8).

Now (*cf.* Convention 2.6.4) choose $M = M_1 > n$ and let $v_{n+1} \equiv w$, $v_{n+2}, \ldots, v_M$ be further program variables of types $k_{n+1} = l$, $k_{n+2}, \ldots, k_M$ such that

$$\mathit{ProgVar}(S) \subseteq \{v_1, \ldots, v_M\}.$$

(Here $M_2 = 0$, since S has no array variables).

Now let $p \equiv p(x_1, \ldots, x_n, y)$ be the following assertion in $\mathit{Lang}_1(\Sigma^*)$:

$$\bigwedge_{i=1}^{n} (x_i \neq \mathsf{unspec}_{k_i}) \wedge$$

$$\wedge \exists Y, X' [\mathsf{Compu}_S(\tilde{X}, Y, X') \wedge \mathsf{Proper}(X') \wedge$$

$$\wedge y = x'_{n+1} \neq \mathsf{unspec}_{k_{n+1}}]$$

(using the notation of Chapter 2, in particular 2.6.11, Definitions 0 and 2), where $\tilde{X}$ is the $(M+1)$-tuple

$$\tilde{X} \equiv (\mathsf{true}, x_1, \ldots, x_n, \mathsf{unspec}_{k_{n+1}}, \ldots, \mathsf{unspec}_{k_M})$$

and Y and X' are tuples of variables

$$Y \equiv (z^{\mathsf{N}}, \xi_0^{\mathsf{B}}, \xi_1, \ldots, \xi_M),$$

$$X' \equiv (x'_0, x'_1, \ldots, x'_M).$$

Note that $\mathit{Var}(p) = \{x_1, \ldots, x_n, y\}$.

Then p expresses the family of partial functions $\langle \psi^A \mid A \in \mathbb{K} \rangle$ over $\mathbb{K}^*$, in the sense that for all $A \in \mathbb{K}$, $\vec{x} \in A[\vec{k}]$ and $\mathbf{y} \in A_l$, if ρ is any valuation over A such that $\rho(x_i) = \mathbf{x}_i$ for $i = 1, \ldots, n$ and $\rho(y) = \mathbf{y}$, then (from Theorem 2 of 2.6.11)

$$A^*, \rho \models p \iff \psi^A(\vec{\mathbf{x}}) \downarrow \mathbf{y}. \qquad \square$$

3.5.3 Representation of $\mathbb{K}^{**}$ in $\mathbb{K}^*$

As an immediate consequence of the theorem in the last subsection, we have that all '**while**' computable functions over $\mathbb{K}^*$ are expressible over $\mathbb{K}^{**}$.

The argument for Step 1 is completed by showing how to code a structure A^{**} within A^*. Now for each sort i, A^{**} has, as domains, $A_i^{\mathsf{u}\mathsf{u}'}$, $A_i^{*\mathsf{u}'}$, $A_i^{\mathsf{u}*}$ and A_i^{**} (where u' is a second unspecified object!).

Consider, for example, the domain A_i^{**}. If $\eta \in A_i^{**}$ (so that $\eta : \mathbb{N} \to (A_i^* \cup \{\mathsf{u}'\})$), then η can be represented by a pair

$$(\beta, \xi) \in \mathbb{B}^* \times A_i^* \qquad (\beta \text{ a "flag"})$$

where (recall the codings in 1.1.7 and 2.6.8)

— if $\eta(m) \in A_i^*$ then $\beta(m) = \mathbb{t}$ and for all n $\xi(<m,n>) = \eta(m)(n)$, and
— if $\eta(m) = \mathsf{u}'$ then $\beta(m) = \mathbb{f}$ and for all n $\xi(<m,n>) = \mathbb{u}$.

Correspondingly, in the assertion language, quantification over domains A_i^{**} can be replaced by quantification over A_i^* (and $\mathbb{B}^*$). Thus, given an assertion p with quantification over A^{**}, but free variables ranging over A^* only, we can find an assertion $\tilde{p}$ in $\boldsymbol{Lang}_1(\Sigma^*)$ such that

$$\mathbb{K} \models p \leftrightarrow \tilde{p} .$$

This completes the argument.

3.6 INDUCTIVE COMPUTABILITY OF THE INPUT-OUTPUT RELATION

3.6.1 Introduction

Recall Convention 2.6.4, namely that all expressions with which we deal contain simple and array variables only among $\vec{v}, \vec{a}$, where

$\vec{v}$ is the tuple $(v_1, \ldots, v_{M_1})$ of sorts $(k_1, \ldots, k_{M_1})$ respectively,
and $\vec{a}$ is the tuple $(a_1, \ldots, a_{M_2})$ of sorts $(l_1, \ldots, l_{M_2})$ respectively.

We again represent, as in Chapter 2, a state over A by a vector of elements of A^*:

$$\boldsymbol{X} = (\boldsymbol{x}_0^{\mathbf{B}}, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{M_1}, \boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_{M_2})$$

(see 2.6.5), with domain

$$\mathbb{B}^{\mathsf{u}} \times A_{k_1}^{\mathsf{u}} \times \cdots \times A_{k_{M_1}} \times A_{l_1}^* \times \cdots \times A_{l_{M_2}}^* ,$$

represented for short as $A^*[\mathsf{B}, \vec{k}, \vec{l}^*]$.

We turn to Step 2 of the plan outlined in 3.5.1. We must show that for all closed programs R, the partial function $\mathcal{M}_A(R)$ is inductively computable over $\mathbb{K}^*$, or, more precisely, there is a tuple of schemes $\vec{\vartheta}_R$, with vectors as arguments and values, such that for all $A \in \mathbb{K}$, if the vectors $\boldsymbol{X}$ and $\boldsymbol{X}'$ represent states σ and σ' over A, then

$$\vec{\vartheta}_R^A(\boldsymbol{X}) \downarrow \boldsymbol{X}' \iff \mathcal{M}_A(R)(\sigma) \downarrow \sigma'$$

where $\vec{\vartheta}_R^A$ is the "realization" of $\vec{\vartheta}_R$ in A^* (see below).

3.6.2 Outline of the notion of inductive computability

A precise definition of the notion of *induction scheme* for $\mathbb{K}$-computability will be given in Section 4.1. Here we merely note that induction schemes are notations for all functions obtained from

(1) the functions named in the signature Σ,

by using the principles of

(2) definition by cases,
(3) composition,
(4) simultaneous primitive recursion on $\mathbb{N}$, and
(5) the least number operator μ over $\mathbb{N}$.

We will systematically point out where these defining principles are used.

With each scheme α can be associated its *type* $\tau(\alpha)=(n;\vec{k},l)$, where $n\geq 0$, $\vec{k}$ is the n-tuple $k_1,\ldots,k_n$, $k_i\in \textbf{\textit{Sort}}(\Sigma)$ for $i=1,\ldots,n$ and $l\in \textbf{\textit{Sort}}(\Sigma)$. Such a scheme α is intended to denote a partial function of type $\tau(\alpha)$. Its semantics is therefore the family of partial functions

$$\{\alpha\}^A:\ A[\vec{k}]\dashrightarrow A_l$$

(where, as before, $A[\vec{k}]=A_{k_1}\times\cdots\times A_{k_n}$) for all $A\in\mathbb{K}$. We will call $\{\alpha\}^A$ the *realization* of α on A.

3.6.3 Coding of syntactic expressions and of states

We assume that we are given a numerical coding, or Gödel numbering, of the set of all syntactic expressions of Σ^* (as in Zucker [1980]), *i.e.* an effective mapping from expressions E to numbers $\ulcorner E\urcorner$, which satisfies certain basic properties:

— $\ulcorner E\urcorner$ increases strictly with $\textbf{\textit{compl}}(E)$, and in particular, the code of an expression is larger than those of its subexpressions;

— sets such as $\{\ulcorner t\urcorner \mid t\in \textbf{\textit{ProgTerm}}\}$, $\{\ulcorner S\urcorner \mid S\in \textbf{\textit{Statemt}}\}$, $\{\ulcorner S\urcorner \mid S$ is an assignment$\}$, etc., are primitive recursive;

— we can go primitive recursively from codes of expressions to codes of their immediate subexpressions, and vice versa; thus, for example, $\ulcorner S_1\urcorner$ and $\ulcorner S_2\urcorner$ are primitive recursive in $\ulcorner S_1;S_2\urcorner$, and conversely, $\ulcorner S_1;S_2\urcorner$ is primitive recursive in $\ulcorner S_1\urcorner$ and $\ulcorner S_2\urcorner$.

In short, *we can primitive recursively simulate all operations involved in processing the syntax of our programming and assertion languages.* This primitive recursiveness will be used in proving inductive computability, by virtue of the theorem in 4.2.1. (Actually a general recursive Gödel

numbering of syntax would be sufficient, for the same reason.)

Furthermore, states σ over A are coded by vectors $\ulcorner\sigma\urcorner$ in A^* as in 2.6.5. In particular, the error state ε is coded by the "error vector"

$$\mathsf{error}^A = (\mathbb{u}_{\mathsf{B}}, \mathbb{u}_{k_1}, \ldots, \mathbb{u}_{k_{M_1}}, \mathsf{Null}^A_{l_1}, \ldots, \mathsf{Null}^A_{l_{M_2}}).$$

3.6.4 Representation of the operational semantics by inductively computable functions

We now give a sequence of four theorems, each of which proves the inductive computability over $\mathbb{K}^*$ of a partial function, or tuple of functions, which simulates some aspect of the operational semantics of the programming language.

We deal in this subsection with induction schemes over $\mathbb{K}^*$ (not $\mathbb{K}$), so by "scheme" we will mean: induction scheme over $\mathbb{K}^*$. (For the effect this has on the schemes of recursion, definition by cases and the μ operator, see 4.1.9—12.)

For any scheme α and $A \in \mathbb{K}$, we will write α^A, instead of the more cumbersome $\{\alpha\}^{A^*}$, for the realization of α in A^*.

We also consider "scheme tuples", *i.e.* n-tuples of induction schemes $\vec{\alpha} \equiv (\alpha_1, \ldots, \alpha_n)$ (for $n \geq 1$), with realizations

$$\vec{\alpha}^A : B_1 \times \cdots \times B_m \dashrightarrow C_1 \times \cdots \times C_n$$

(each B_j and C_k being some $A_i^{\mathbb{u}}$ or A_i^*). Such a realization is an n-tuple of partial functions, whose components are the individual realizations

$$\alpha_j^A : B_1 \times \cdots \times B_m \dashrightarrow C_j \qquad (j = 1, \ldots, n).$$

For ease of exposition, we will not give the precise derivations of the induction schemes. We trust that our semi-formal approach will be convincing to the reader.

THEOREM 1 (Program term evaluation). *For each sort k of Σ there is a scheme φ_k, where*

$$\varphi_k^A : \mathbb{N}^{\mathbb{u}} \times A^*[\mathsf{B}, \vec{k}, \vec{l}^*] \to A_k^{\mathbb{u}},$$

which represents the evaluation function $\mathcal{R}$ for program terms of 1.2.5, in the sense that, for every $t \in \mathbf{ProgTerm}_k$, $A \in \mathbb{K}$ and $\sigma \in \mathrm{PR.STATE}(A)$,

$$\varphi_k^A(\ulcorner t \urcorner, \ulcorner \sigma \urcorner) = \mathcal{R}_A(t)(\sigma). \tag{1}$$

PROOF. The family $\langle \varphi_k \mid k \in \mathbf{Sort} \rangle$ is defined by simultaneous recursion; that is, $\varphi_k(n, X)$ is defined by simultaneous (over k) course-of-values

recursion on n. The definition is straightforward (*cf.* Definition A.10 in Zucker [1980]).

Let $X \equiv (x_0^{\mathsf{B}}, x_1, \ldots, x_{M_1}, \xi_1, \ldots, \xi_{M_2})$, with x_i of type k_i and ξ_j of type l_j. Then (recalling Convention 2.6.4)

$$\varphi_k(z, X) = \mathbb{u}_k \quad \text{if } \mathsf{Unspec}_{\mathsf{N}}(z) \text{ or } z \text{ is } \textit{not} \text{ the code of a program term of sort } k, \text{ with program variables among } \vec{v}, \vec{a} \text{ only.}$$

Otherwise, paralleling the definition of $\mathcal{R}_A$, we define, for $i = 1, \ldots, M_1$ and $j = 1, \ldots, M_2$:

$$\varphi_{k_i}(\ulcorner v_i \urcorner, X) = x_i$$

$$\varphi_{l_j}(\ulcorner a_j[t] \urcorner, X) \simeq \mathsf{Ap}^{\mathsf{u}}_{l_j}(\xi_j, \varphi_{\mathsf{N}}(\ulcorner t \urcorner, X))$$

$$\varphi_l(\ulcorner F(t_1, \ldots, t_m) \urcorner, X) = F^{\mathsf{u}}(\varphi_{k_1}(\ulcorner t_1 \urcorner, X), \ldots, \varphi_{k_m}(\ulcorner t_m \urcorner, X))$$

where F has type $(m; k_1, \ldots, k_m, l)$, and finally:

$$\varphi_k(\ulcorner \textsf{if } b \textsf{ then } t_1 \textsf{ else } t_2 \textsf{ fi} \urcorner, X) = \begin{cases} \varphi_k(\ulcorner t_1 \urcorner, X) & \text{if } \varphi_{\mathsf{B}}(\ulcorner b \urcorner, X) \downarrow \mathbb{t} \\ \varphi_k(\ulcorner t_2 \urcorner, X) & \text{if } \varphi_{\mathsf{B}}(\ulcorner b \urcorner, X) \downarrow \mathbb{f} \\ \mathbb{u}_k & \text{if } \varphi_{\mathsf{B}}(\ulcorner b \urcorner, X) \downarrow \mathbb{u}. \end{cases}$$

It follows from Corollary 2 in 4.5.5 that the tuple $\vec{\varphi} \equiv \langle \varphi_k \mid k \in \textbf{\textit{Sort}} \rangle$ is defined by simultaneous course-of-values recursion, of the form

$$\varphi_k(n, X) \simeq \chi_k(n, X, \vec{\varphi}(\delta_1(n), X), \ldots, \vec{\varphi}(\delta_m(n), X))$$

Here, if n is the code of a program term (other than a simple variable or constant), then $\delta_1(n), \ldots, \delta_m(n)$ are codes of its immediate subterms (which are less than n, by 3.6.3), with $\delta_1, \ldots, \delta_m$ primitive recursive, and hence inductively computable (4.1.1); and χ_k is a definition by cases, with the different cases primitive recursively decidable, and hence again inductively computable.

This course-of-values recursion is of the kind reducible to primitive recursion, using the type k^* to code finite sequences of the range type k, by a standard technique (see 4.5.7).

Now equation (1) can be proved by induction on $\textbf{\textit{compl}}(t)$. □

Notice that φ_k is a *total function, i.e.* for all $A \in \mathbb{K}$, all n and all vectors $\boldsymbol{X}$ in A^*, $\varphi_k^A(n, \boldsymbol{X}) \downarrow$ (proved by induction on n).

THEOREM 2 (Computation step). *There is a scheme tuple* $\vec{\psi}$, *with*

$$\vec{\psi}^A : \mathbb{N}^u \times \mathbb{N}^u \times A^*[\mathsf{B}, \vec{k}, \vec{l}^*] \to \mathbb{B}^u \times A^*[\mathsf{B}, \vec{k}, \vec{l}^*]$$

which represents the function ***CompStep*** *of 3.1.5, in the sense that for every* $n \in \mathbb{N}$, $R \in$ ***ClProg***, $A \in \mathbb{K}$ *and* $\sigma \in \mathrm{PR.STATE}(A)$,

$$\vec{\psi}^A(n, \ulcorner R \urcorner, \ulcorner \sigma \urcorner) \simeq \begin{cases} (\mathbb{t}, \ulcorner \sigma' \urcorner) & \text{if } \textbf{\textit{CompStep}}_A(n)(R)(\sigma) = \sigma' \\ & (\text{where } \sigma' \text{ may be } \varepsilon) \\ (\mathbb{f}, \mathsf{error}^A) & \text{if } \textbf{\textit{CompStep}}_A(n)(R)(\sigma) = *. \end{cases} \tag{2}$$

PROOF. We will actually follow the definition of $\textbf{\textit{CompStep}}_A$ in 3.1.6, which avoids course-of-values recursion. To begin with, we must define schemes α and β with

$$\alpha, \beta : \mathbb{N}^u \times A^*[\mathsf{B}, \vec{k}, \vec{l}^*] \to \mathbb{N}^u$$

which represent the functions ***first*** and ***rest*** of 3.1.6, in the sense that for every $R \in$ ***ClProg***, $A \in \mathbb{K}$ and $\sigma \in \mathrm{PR.STATE}(A)$,

$$\begin{aligned} \alpha^A(\ulcorner R \urcorner, \ulcorner \sigma \urcorner) &= \ulcorner \textbf{\textit{first}}_A(R)(\sigma) \urcorner \\ \beta^A(\ulcorner R \urcorner, \ulcorner \sigma \urcorner) &= \ulcorner \textbf{\textit{rest}}_A(R)(\sigma) \urcorner. \end{aligned} \tag{3}$$

The definition of α is given by

$$\alpha(z, X) = \mathbb{u}_{\mathbb{N}} \text{ if } \mathsf{Unspec}_{\mathbb{N}}(z) \text{ or } z \text{ is } \textit{not} \text{ the code of a closed program with program variables among } \vec{v}, \vec{a} \text{ only.}$$

Otherwise (paralleling the definition of $\textbf{\textit{first}}_A$):

$$\alpha(\ulcorner < D \mid S > \urcorner, X) = \ulcorner S \urcorner \quad \text{if } S \text{ is atomic}$$

$$\alpha(\ulcorner < D \mid S_1; S_2 > \urcorner, X) \simeq \alpha(\ulcorner < D \mid S_1 > \urcorner, X)$$

etc., etc. (as in 3.1.6). Similarly for the definition of β.

Again, α and β are defined by course-of-values recursion, of a kind reducible to primitive recursion (see 4.5.7).

Equation (3) is then proved by induction on $\textbf{\textit{compl}}(R)$.

Notice that α and β are total functions.

Next we define a scheme tuple $\vec{\gamma}$, with

$$\vec{\gamma}^A : \mathbb{N}^u \times A^*[\mathsf{B}, \vec{k}, \vec{l}^*] \to A^*[\mathsf{B}, \vec{k}, \vec{l}^*],$$

to represent the meaning function $\mathcal{M}$ of Chapter 1, in the sense that for every atomic S, $A \in \mathbb{K}$ and $\sigma \in \mathrm{PR.STATE}(A)$,

$$\vec{\gamma}(\ulcorner S \urcorner, \ulcorner \sigma \urcorner) = \ulcorner \mathcal{M}_A(S)(\sigma) \urcorner. \tag{4}$$

The definition of $\vec{\gamma}$ is a simple definition by cases:

$$\vec{\gamma}(z, X) = \mathsf{error} \quad \text{if } \mathsf{Unspec}_{\mathsf{N}}(z) \text{ or } z \text{ is } \textit{not} \text{ the code of an atomic statement with program variables among } \vec{v}, \vec{a} \text{ only.}$$

Otherwise (paralleling the definition of $\mathcal{M}_A$ in 1.2.11):

$$\vec{\gamma}(\ulcorner \mathsf{skip} \urcorner, X) = X$$

$$\vec{\gamma}(\ulcorner \mathsf{abort} \urcorner, X) = \mathsf{error}$$

$$\vec{\gamma}(\ulcorner v_i := t \urcorner, X) \simeq \begin{cases} \mathsf{error} & \text{if } \varphi_{k_i}(\ulcorner t \urcorner, X) \downarrow \mathbb{u}_{k_i} \\ X' & \text{if } \varphi_{k_i}(\ulcorner t \urcorner, X) \downarrow \neq \mathbb{u}_{k_i} \end{cases}$$

where X' is the vector with the same components as X, except that x_i is replaced by $\varphi_{k_i}(\ulcorner t \urcorner, X)$.

Note that this case involves a definition by cases, where the boolean test has the form: $\mathsf{Unspec}_{k_i}(\varphi_{k_i}(\ulcorner t \urcorner, X))$.

Finally:

$$\vec{\gamma}(\ulcorner a_j[t_0] := t \urcorner, X) \simeq \begin{cases} \mathsf{error} & \text{if } \varphi_{\mathsf{N}}(\ulcorner t_0 \urcorner, X) \downarrow \mathbb{u}_{\mathsf{N}} \text{ or } \varphi_{l_j}(\ulcorner t \urcorner, X) \downarrow \mathbb{u}_{l_j} \\ X' & \text{if } \varphi_{\mathsf{N}}(\ulcorner t_0 \urcorner, X) \downarrow \neq \mathbb{u}_{\mathsf{N}} \text{ and } \varphi_{l_j}(\ulcorner t \urcorner, X) \downarrow \neq \mathbb{u}_{l_j} \end{cases}$$

where X' is the vector with the same components as X, except that ξ_j is replaced by

$$\mathsf{Adjoin}_{l_j}(\xi_j, \varphi_{\mathsf{N}}(\ulcorner t_0 \urcorner, X), \varphi_{l_j}(\ulcorner t \urcorner, X)).$$

Note that this case also involves a definition by cases, where the boolean test is the disjunction of $\mathsf{Unspec}_{\mathsf{N}}(\varphi_{\mathsf{N}}(\ulcorner t_0 \urcorner, X)$ and $\mathsf{Unspec}_{l_j}(\varphi_{l_j}(\ulcorner t \urcorner, X))$ (disjunction being derived from the logical functions included in the signature).

Equation (4) now follows directly from (1).

Notice again that the functions $\vec{\gamma}$ are total.

We turn finally to the definition of $\vec{\psi}$, as required by the theorem. We will use the notation

$$\mathsf{star} \equiv_{df} (\mathbb{1}, \mathsf{error})$$

and let κ be a scheme for concatenating (codes of) declarations with statements, *i.e.* such that

$$\kappa^A(\ulcorner D\urcorner, \ulcorner S\urcorner) = \ulcorner < D \mid S > \urcorner$$

(which is possible, since this operation is primitive recursive, *cf.* 4.2.1).

The tuple $\vec{\psi}$ is then defined as follows.

$$\dot{\psi}(z_1, z_2, X) = \textsf{star} \quad \text{if } \textsf{Unspec}_{\textsf{N}}(z_1) \text{ or } \textsf{Unspec}_{\textsf{N}}(z_2) \text{ or } z_2 \text{ is } \textit{not} \text{ the code of a of a closed program with program variables among } \vec{v}, \vec{a} \text{ only.}$$

Otherwise (paralleling the definition of $\textbf{\textit{CompStep}}_A$ in 3.1.6): for $n = 0$, define

$$\vec{\psi}(0, \ulcorner < D \mid S > \urcorner, X) = (\Bbbk, X).$$

Next (for $n > 0$) put

$$X' \equiv (x_0', \ldots) =_{df} \vec{\gamma}(\alpha(\ulcorner < D \mid S > \urcorner, X), X).$$

Then $$\vec{\psi}(1, \ulcorner < D \mid S > \urcorner, X) \simeq (\Bbbk, X')$$

and for $n > 1$,

$$\dot{\psi}(n, \ulcorner < D \mid S > \urcorner, X) \simeq \begin{cases} \vec{\psi}(n-1, \kappa(\ulcorner D \urcorner, \beta(\ulcorner < D \mid S > \urcorner, X)), X') \\ \qquad \text{if } S \text{ is not atomic and } x_0' \equiv \textsf{true}, \\ \textsf{star} \quad \text{otherwise.} \end{cases}$$

The scheme $\vec{\psi}$ is again definable by (simultaneous) primitive recursion (4.5.7).

Now (2) can be proved by induction on $(n, \textbf{\textit{compl}}(R))$, using (3). □

Notice, again, that $\vec{\psi}$ is a tuple of total functions.

THEOREM 3 (Length of computation). *There is a scheme η, with*

$$\eta^A : \mathbb{N}^{\mathsf{u}} \times A^*[\mathsf{B}, \vec{k}, \vec{l}^*] \dashrightarrow \mathbb{N}^{\mathsf{u}},$$

which represents the function $\textbf{\textit{LengthComp}}$ *of 3.1.5, in the sense that for every* $R \in \textbf{\textit{ClProg}}$, $A \in \mathbb{K}$ *and* $\sigma \in \text{PR.STATE}(A)$,

$$\eta(\ulcorner R \urcorner, \ulcorner \sigma \urcorner) \simeq \begin{cases} \textbf{\textit{LengthComp}}_A(R)(\sigma) & \textit{if } \textbf{\textit{LengthComp}}_A(R)(\sigma) < \infty \\ \uparrow & \textit{otherwise}. \end{cases}$$

PROOF. Consider the scheme tuple of Theorem 2,

$$\vec{\psi} \equiv (\psi_{-1}, \psi_0, \psi_1, \ldots, \psi_{M_1 + M_2}),$$

where the first component is the boolean-valued function

$$\psi_{-1} : \mathbb{N}^{\mathsf{u}} \times \mathbb{N}^{\mathsf{u}} \times A^*[\mathsf{B}, \vec{k}, \vec{l}^*] \to \mathbb{B}^{\mathsf{u}}$$

such that $\psi_{-1}^{A}(n, \ulcorner R \urcorner, \ulcorner \sigma \urcorner) = \mathbb{t}$ if the computation of R starting at σ has not been completed at step n, and $= \mathbb{f}$ otherwise.

Now simply define

$$\eta(m, X) \simeq \mu n[\mathsf{not}(\psi_{-1}(n, m, X))]. \qquad \square$$

Notice the use of the μ operator in the above proof.

Finally we have:

THEOREM 4 (Universal program evaluation). *There is a scheme tuple* $\vec{\vartheta}$, *with*

$$\vec{\vartheta}^{A} : \mathbb{N}^{\mathsf{u}} \times A^*[\mathsf{B}, \vec{k}, \vec{l}^*] \dashrightarrow A^*[\mathsf{B}, \vec{k}, \vec{l}^*],$$

which represents the meaning function $\mathcal{M}\,(=O)$ *for programs, in the sense that for every* $R \in \mathbf{ClProg}$, $A \in \mathbb{K}$, $\sigma \in \mathrm{PR.STATE}(A)$ *and* $\sigma' \in \mathrm{STATE}(A)$,

$$\vec{\vartheta}^{A}(\ulcorner R \urcorner, \ulcorner \sigma \urcorner) \simeq \begin{cases} \ulcorner \sigma' \urcorner & \text{if } \mathcal{M}_A(R)(\sigma) \downarrow \sigma' \\ \uparrow & \text{if } \mathcal{M}_A(R)(\sigma) \uparrow. \end{cases}$$

PROOF. Consider again the scheme tuple of Theorem 2,

$$\vec{\psi} \equiv (\psi_{-1}, \psi_0, \psi_1, \ldots, \psi_{M_1+M_2}).$$

Let $\vec{\psi}'$ be the tuple without ψ_{-1}, *i.e.*

$$\vec{\psi}' \equiv (\psi_0, \psi_1, \ldots, \psi_{M_1+M_2}).$$

Now simply define

$$\vec{\vartheta}(m, X) \simeq \vec{\psi}'(\eta(m, X), m, X). \qquad \square$$

Notice that $\vec{\vartheta}$ is *not* total; in fact (by the totality of $\vec{\psi}$ and the convention for composing partial functions, *cf.* Section 2.1(4)) $\vec{\vartheta}(m, X)$ is defined precisely when $\eta(m, X)$ is.

3.6.5 Conclusion: Expressibility of weakest precondition and strongest postcondition

Let us see what we have achieved. Consider again the three steps of the plan outlined in 3.5.1.

(*Step 2.*) It follows from the work in this section that for any closed program R, $\mathcal{M}_A(R)$ is inductively computable over $\mathbb{K}^*$ by the scheme tuple $\vec{\vartheta}_R$, defined by

$$\vec{\vartheta}_R(X) \simeq \vec{\vartheta}(\ulcorner R \urcorner, X).$$

(*Step 3.*) It will be shown in Section 4.3 that every inductively computable function is 'while' computable. Hence $\mathcal{M}_A(R)$ is 'while' computable over $\mathbb{K}^*$.

(*Step 1.*) It was shown (3.5.2/3) that every 'while' computable function over $\mathbb{K}^*$ is expressible over $\mathbb{K}^*$. Hence, in particular, there is an assertion $\mathsf{io}_R(X, X')$ of the assertion language $\boldsymbol{Lang}_1(\Sigma^*)$ which expresses the function representing $\mathcal{M}_A(R)$, uniformly for $A \in \mathbb{K}$, in the sense that for all A, ρ, σ and σ': if σ and σ' represent X and X' respectively relative to ρ, then

$$A^*, \rho \models \mathsf{io}_R(X, X') \iff \sigma \neq \varepsilon \text{ and } \mathcal{M}_A(R)(\sigma) \downarrow \sigma'. \qquad (1)$$

Finally we can define assertions expressing the weakest precondition and strongest postcondition as in 3.5.1, namely

$$\mathsf{wp}_{ra}[R, p] \equiv_{df} \forall X, X'[\mathsf{Val}(X, \vec{v}, \vec{a}) \wedge \mathsf{io}_R(X, X') \supset \supset \mathsf{Proper}(X') \wedge p[\![X']\!]]$$

and $$\mathsf{sp}_{ra}[p, R] \equiv_{df} \exists X', X[p[\![X]\!] \wedge \mathsf{io}_R(X', X) \wedge \mathsf{Val}(X, \vec{v}, \vec{a})]$$

(with $p[\![X']\!]$ as given in 2.6.7, Definition 2, and Proper and Val as in 2.6.11, Definitions 0 and 3).

These satisfy the same properties as the corresponding assertions in Chapter 2, *viz.* 2.6.11, Theorems 4 and 5 and their Corollaries. (The proofs are the same, except that we now use the assertion io_R with property (1) above, instead of the assertion Compu_S of Chapter 2.)

3.6.6 Functions computable by 'while'-array programs and functions computable by recursive programs: Two conjectures

In the last two sections an important part was played by functions computable by 'while' programs over $\mathbb{K}^*$. Although this was largely for technical reasons (in order to prove expressibility of the weakest precondition for $\boldsymbol{ProgLang}_{ra}$ over $\boldsymbol{Lang}_1(\Sigma^*)$), the class of 'while' programs over $\mathbb{K}^*$ is nevertheless an interesting model of computation in its own right. The elements of the domains A_i^* are, essentially, arrays. Hence an assignment to a variable of type i^* can be viewed as an (unsubscripted) *array assignment*. It is then a simple exercise to interpret the class of 'while' programs with arrays over $\mathbb{K}$ within the class of (arrayless) 'while' programs over $\mathbb{K}^*$: for example, the assignment $a[t_0] := t$ would be translated as $a := \mathsf{Adjoin}_i(a, t_0, t)$.

We conjecture that the converse is also true, in the sense that the class of '**while**' *computable* functions over $\mathbb{K}^*$ (with arguments and values in the "unstarred" types only) is equivalent to the class of '**while**'-*array computable* functions over $\mathbb{K}$ (see 4.3.1—4.3.3, 4.6.1 for exact definitions).

This conjecture can be reformulated in the language of inductive definability over $\mathbb{K}$ and $\mathbb{K}^*$ (by the results of Chapter 4).

We further conjecture that either of these classes is equal to the class of functions *computable by recursive programs* (without arrays) over $\mathbb{K}$.

3.7 COMPLETENESS OF THE PROOF SYSTEM

We adapt the completeness proof of Gorelick [1975]. We begin with a key lemma (*cf.* [dB], Lemma 5.8, or Apt [1981], §3.6).

LEMMA 1. *Let* $D \equiv \langle P_i \Leftarrow S_i \rangle_{i=1}^{m}$ *be a closed declaration (with* $\mathbf{ProgVar}(D) \subseteq \vec{v}, \vec{a}$, *as in Convention 2.6.4), and for* $i = 1, \ldots, m$ *let*

$$r_i \equiv \mathsf{sp}_{ra}[\mathsf{Val}(X, \vec{v}, \vec{a}), \langle D \mid P_i \rangle],$$

and

$$f_i \equiv \{\mathsf{Val}(X, \vec{v}, \vec{a})\} P_i \{r_i\},$$

and let $\vec{f} \equiv f_1, \ldots, f_m$. *Then for any* S, p, q *such that* $\langle D \mid S \rangle$ *is a closed program (and* $\mathbf{ProgVar}(S, p, q) \subseteq \vec{v}, \vec{a}$*),*

if $$\mathbb{K} \models \langle D \mid \{p\} S \{q\} \rangle$$

then $$\mathbb{K} \vdash \langle D \mid \vec{f} \rightarrowtriangle \{p\} S \{q\} \rangle.$$

(*Note.* The assertions $\mathsf{sp}_{ra}[\ ,\ ]$ and $\mathsf{Val}(\ )$ were defined in 3.6.5 and 2.6.11, Definition 3, respectively.)

PROOF. By induction on $\mathbf{compl}(S)$. For S not a procedure variable, the argument is similar to that in 1.7.1. Consider now the case $S \equiv P_i$ for some i $(1 \leq i \leq m)$. Assume that

$$\mathbb{K} \models \langle D \mid \{p\} P_i \{q\} \rangle. \tag{1}$$

Let $p_1 \equiv p\langle X''/X \rangle$ and $q_1 \equiv q\langle X''/X \rangle$, where X'' is a tuple of fresh variables (disjoint from X, and not occurring free or bound in p or q), and '$\langle X''/X \rangle$' denotes simultaneous substitution (which is the same here as repeated substitution, since X'' is disjoint from X).

Now (arguing in $\mathbf{ProofSys}_{ra}$ (3.4.1) but omitting '$\mathbb{K} \vdash$' and 'D'), by the selection rule

$$\vec{f} \rightarrowtriangle \{\mathsf{Val}(X, \vec{v}, \vec{a})\} P_i \{r_i\},$$

so by the invariance rule

$$\vec{f} \twoheadrightarrow \{\mathsf{Val}(X,\vec{v},\vec{a}) \wedge p_1[\![X]\!]\}P_i\{r_i \wedge p_1[\![X]\!]\}, \tag{2}$$

since $p_1[\![X]\!]$ contains no program variables (*cf.* 2.6.7, Definition 2 and Remark (2) following).

Next we show

$$\mathbb{K} \models r_i \wedge p_1[\![X]\!] \supset q_1. \tag{3}$$

Assume, for given $I=(A^*,\rho,\sigma)$,

$$I \models r_i \wedge p_1[\![X]\!].$$

So $$I \models r_i \tag{4}$$

and $$I \models p_1[\![X]\!]. \tag{5}$$

By (4) and (the analogue of) 2.6.11, Theorem 5, there is a state $\sigma' \neq \varepsilon$ such that

$$\rho, \sigma' \models \mathsf{Val}(X,\vec{v},\vec{a}) \tag{6}$$

and $$\mathfrak{M}_A(D \mid P_i)(\sigma') = \sigma. \tag{7}$$

By (6) and 2.6.11, Theorem 3,

$$X \text{ represents } \sigma' \text{ relative to } \rho. \tag{8}$$

By (5), (8) and the theorem in 2.6.7,

$$\rho, \sigma' \models p_1 \tag{9}$$

(since $\boldsymbol{Var}(p_1) \cap X = \emptyset$). By (1) and the soundness of the substitution rule (repeatedly applied),

$$\mathbb{K} \models <D \mid \{p_1\}P_i\{q_1\}>.$$

Hence by (9) and (7), $$I \models q_1.$$

This proves (3). Hence by (2), the oracle rule applied to (3) and the consequence rule,

$$\vec{f} \twoheadrightarrow \{\mathsf{Val}(X,\vec{v},\vec{a}) \wedge p_1[\![X]\!]\}P_i\{q_1\}.$$

By repeated application of the existential quantification rule,

$$\vec{f} \twoheadrightarrow \{\exists X\,[\mathsf{Val}(X,\vec{v},\vec{a}) \wedge p_1[\![X]\!]]\}P_i\{q_1\} \tag{10}$$

(since $\boldsymbol{Var}(q_1) \cap X = \emptyset$). Now by the Corollary of Theorem 3 in 2.6.11,

$$\mathbb{K} \models p_1 \supset \exists X\,[\mathsf{Val}(X,\vec{v},\vec{a}) \wedge p_1[\![X]\!]]$$

(since $\mathbf{Var}(p_1) \cap X = \emptyset$), so by the oracle rule, (10) and the consequence rule,

$$\vec{f} \rightarrowtriangle \{p_1\} P_i \{q_1\}.$$

Then by repeated applications of the substitution rule,

$$\vec{f} \rightarrowtriangle \{p_1 < X/X'' >\} P_i \{q_1 < X/X'' >\}. \tag{11}$$

Now (since the variables in X'' do not occur free or bound in p or q) $p_1 < X/X'' > \equiv p$ and $q_1 < X/X'' > \equiv q$, so (11) is identical to

$$\vec{f} \rightarrowtriangle \{p\} P_i \{q\}. \qquad \square$$

LEMMA 2. *With the notation of Lemma 1:*

$$\mathbb{K} \vdash < D \mid \{\mathsf{Val}(X, \vec{v}, \vec{a})\} P_i \{r_i\} > \qquad \textit{for } i = 1, \ldots, m.$$

PROOF. Since

$$\mathbb{K} \models < D \mid \{\mathsf{Val}(X, \vec{v}, \vec{a})\} P_i \{r_i\} >$$

and $\mathcal{M}_A(S_i) = \mathcal{M}_A(P_i)$ for all $A \in \mathbb{K}$, therefore

$$\mathbb{K} \models < D \mid \{\mathsf{Val}(X, \vec{v}, \vec{a})\} S_i \{r_i\} > \qquad \text{for } i = 1, \ldots, m.$$

So by Lemma 1,

$$\mathbb{K} \vdash < D \mid \vec{f} \rightarrowtriangle \{\mathsf{Val}(X, \vec{v}, \vec{a})\} S_i \{r_i\} > \qquad \text{for } i = 1, \ldots, m,$$

where $\vec{f} \equiv f_1, \ldots, f_m$ with $f_i \equiv \{\mathsf{Val}(X, \vec{v}, \vec{a})\} P_i \{r_i\}$. The result follows by the recursion rule. $\square$

REMARK (*Freezing of variables*). Note that (by the definitions of Val in 2.6.11, Definition 3, and $p[\![X]\!]$ in 2.6.7, Definition 2) the assertion $\mathsf{Val}(X, \vec{v}, \vec{a})[\![X']\!]$ is equivalent to $X = X'$, *i.e.*

$$\mathbb{K} \models \mathsf{Val}(X, \vec{v}, \vec{a})[\![X']\!] \leftrightarrow X = X'.$$

Hence (by the definition of sp_{ra} in 3.6.5) r_i is equivalent to

$$\exists X' [\mathsf{io}_{< D \mid P_i >}(X, X') \wedge \mathsf{Val}(X', \vec{v}, \vec{a})].$$

Thus, rewriting f_i as

$$\{\mathsf{Val}(X, \vec{v}, \vec{a})\} P_i \{\exists X' [\mathsf{io}_{< D \mid P_i >}(X, X') \wedge \mathsf{Val}(X', \vec{v}, \vec{a})]\},$$

we see that in the above proofs the tuple of assertion variables X was used to "freeze" the values of the program variables $\vec{v}, \vec{a}$ before the procedure calls.

We come now to the completeness theorem. Note that it only applies to sequents of the form $< D \mid \{p\}S\{q\} >$, *i.e.* without antecedents.

THEOREM (Completeness of $\boldsymbol{ProofSys}_{ra}(\mathbb{K})$).

If $\mathbb{K} \models < D \mid \{p\}S\{q\} >$

then $\mathbb{K} \vdash < D \mid \{p\}S\{q\} >$.

PROOF. Suppose $\models < D \mid \{p\}S\{q\} >$. Now choose $\vec{v}$ and $\vec{a}$ so that $\boldsymbol{ProgVar}(D,S,p,q) \subseteq \vec{v},\vec{a}$. By Lemma 1,

$$\mathbb{K} \vdash < D \mid \vec{f} \rightarrowtriangle \{p\}S\{q\} > \qquad (1)$$

where $\vec{f} \equiv f_1, \ldots, f_m$ with $f_i \equiv \{\mathsf{Val}(X,\vec{v},\vec{a})\}P_i\{r_i\}$. By Lemma 2,

$$\mathbb{K} \vdash < D \mid f_i > \qquad \text{for } i = 1, \ldots, m. \quad (2)$$

From (1) and (2) we obtain, by repeated applications of the cut rule:

$$\mathbb{K} \vdash < D \mid \{p\}S\{q\} >. \qquad \square$$

3.8 APPENDIX: TOTAL CORRECTNESS FOR RECURSIVE PROGRAMS

We restrict ourselves to some remarks.

The simplest semantics for total correctness does not involve n-satisfaction or n-validity (as in 3.3.2). Satisfaction of declared correctness formulae can be defined as in 2.8.1: for $I = (A^*, \rho, \sigma)$, $I \models^T < D \mid \{p\}S\{q\} >$ iff

$$A^*, \rho, \sigma \models p \;\Rightarrow$$
$$\text{for some } \sigma' \neq \varepsilon,\ \mathcal{M}_A(D \mid S)(\sigma) \downarrow \sigma' \text{ and } A^*, \rho, \sigma' \models q\,.$$

Then satisfaction of sequents can be defined by:

$$I \models^T < D \mid \vec{f} \rightarrowtriangle f > \quad \text{iff} \quad (I \models^T < D \mid \vec{f} > \;\Rightarrow\; I \models^T < D \mid f >),$$

and total $\mathbb{K}$-validity by:

$$\mathbb{K} \models^T g \quad \text{iff} \quad I \models^T g \text{ for all } I \in \mathrm{INTERP}^*(\mathbb{K}).$$

(Compare the concluding remark in 3.3.3.)

A proof system for total correctness of recursive programs can be obtained from $\boldsymbol{ProofSys}_{ra}$ (3.4.1) by removing the rules for substitution and existential quantification, and replacing the recursion rule (A.5) by:

$(A.5)^T$ *Recursion for total correctness*:

Suppose $D \equiv < P_i \Leftarrow S_i >_{i=1}^m$. Then we have the following m rules, all with the same $2m$ premisses:

$$\frac{< D \mid \vec{f}_0, \{\exists z < n[r_i(z)]\} P_i \{q_i\}_{i=1}^m \rightarrow \{r_j(n)\} S_j \{q_j\} > \quad \text{for } j=1,\ldots,m,}{< D \mid \vec{f}_0 \rightarrow p_j \supset \exists n\,[r_j(n)] > \quad \text{for } j=1,\ldots,m}$$
$$< D \mid \vec{f}_0 \rightarrow \{p_i\} P_i \{q_i\} >$$

for $i=1,\ldots,m$, where $\vec{f}_0$ is some vector of correctness formulae, and for $j=1,\ldots,m$, $r_j(z)$ is an assertion with the free assertion variable z of type N (not free in $\vec{f}_0, p_j$ or q_j), and n and k range over natural numbers (as in 2.8.2). Thus, for example, '$r_j(n)$' stands for '$z \neq \mathsf{unspec}_{\mathsf{N}} \wedge r_j(z)$' (and, of course, '$\{\exists z < n[r_i(z)]\} P_i \{q_i\}_{i=1}^m$' stands for the vector '$\{\exists z < n[r_1(z)]\} P_1 \{q_1\}, \ldots, \{\exists z < n[r_m(z)]\} P_m \{q_m\}$').

Versions of this rule were given in Apt [1981] and Sokolowski [1977]. (See also the references in Apt [1981], §3.9).

It is not hard to show that this rule is sound, according to the above semantics.

The notion of *weakest precondition for total correctness* is defined exactly as in 2.8.3, and is $\mathbb{K}$-expressible by the assertion

$$\mathsf{wp}^T_{ra}[R,p] \equiv_{df} \exists X, X'[\mathsf{Val}(X,\vec{v},\vec{a}) \wedge \mathsf{io}_R(X,X') \wedge \wedge \mathsf{Proper}(X') \wedge p[\![X']\!]]$$

(compare the definition of $\mathsf{wp}_{ra}[R,p]$ in 3.5.1). This satisfies (analogues of) the properties listed for wp^T_{iva} in 2.8.3.

Again, the strongest postcondition for total correctness may be defined exactly as for partial correctness (3.5.1).

We come now to the issue of a completeness proof for total correctness, and encounter the following problem: the substitution and existential quantification rules (3.4.1, C.3 and C.4) are not valid under the above semantics — that is, unless we make the additional restriction that $x \notin \mathit{Var}(\vec{f})$ (in the notation of 3.4.1). However if we try to adapt the proof in Apt [1981], §3.9, to our formalism, we seem to need the existential quantification rule without this restriction.

If, on the other hand, we use the following semantics for validity of sequents (similar to that in 3.3.2):

$$\mathbb{K} \models^T < D \mid f > \quad \text{iff} \quad I \models^T < D \mid f > \text{ for all } I \in \mathrm{INTERP}^*(\mathbb{K}),$$

and $\mathbb{K} \models^T < D \mid \vec{f} \rightarrow f >$ iff $(\mathbb{K} \models^T < D \mid \vec{f} > \Rightarrow \mathbb{K} \models^T < D \mid f >)$,

then, although the substitution and existential quantification rules become

valid, the recursion rule $(A.5)^T$ above is no longer valid. (The problem is that the value of the number variable n is no longer preserved across the '$\rightarrow\!\triangleright$', since such values are quantified separately on each side.)

A solution to this problem is given by America and De Boer [1987]. The idea, briefly, is to define a subsort ***Count*** of the sort **N**. Variables of sort ***Count*** are (roughly) subject to the first semantics, and all other variables to the second. The variable n in the recursion rule must be of sort ***Count***, and the variable x in the substitution and existential quantification rules must *not* be of sort ***Count***.

That this modification yields a complete system is not obvious: in the completeness proof as presented in Apt [1981], the existential quantification rule is applied to a variable which is used in the recursion rule. In America and De Boer [1987], it is shown how to eliminate this particular application of the elimination rule in the proof of the completeness theorem.

Chapter 4

Computability in an Abstract Setting

4.1 INDUCTION SCHEMES

We begin our study of the computability of functions over the class $\mathbb{K}$ by looking at mechanisms for defining functions on a single structure A.

4.1.1 Functions computable by induction on A

Imagine the set of partial functions on A built up from its basic operations, and some simple functions like projections, by means of *composition*; (*simultaneous*) *primitive recursion* on the standard numerical domain $\mathbb{N}$ of A, with algebraic parameters; and the *least number search operator* over $\mathbb{N}$, with algebraic parameters. These functions we will call the *inductively definable* or *inductively computable partial functions* on A; clearly, they are a modest abstraction of the partial recursive functions on $\mathbb{N}$, based on the latter's definition by function schemes. We will describe the set IND(A) of all inductively definable functions on A, following the style of Kleene [1952], §43. First we establish some important notations.

Let $\vec{k}=(k_1,\ldots,k_n)$ and $\vec{l}=(l_1,\ldots,l_m)$ be arbitrary lists of sorts of A.

Let $A[\vec{k}]$ denote $A_{k_1}\times\cdots\times A_{k_n}$ and $A[\vec{l}]$ denote $A_{l_1}\times\cdots\times A_{l_m}$.

Let $\vec{x}=(x_1,\ldots,x_n)$ denote an arbitrary vector of $A[\vec{k}]$ and $\vec{y}=(y_1,\ldots,y_m)$ denote an arbitrary vector of $A[\vec{l}]$.

Let k and l also denote sorts of A. Let $x,y\in A_k$, $z\in\mathbb{N}$ and $b\in\mathbb{B}$.

In this chapter, we sometimes use '$\rightharpoonup$' for partial functions (*cf.* 2.1).

DEFINITION. The set IND(A) consists of partial functions on A of the form $f: A[\vec{k}]\rightharpoonup A_l$. The set is defined inductively by six clauses in which each equation, or system of equations, defines such a partial function f on A of the indicated type.

BASIC FUNCTIONS

I. *Primitive operations of* A. For each primitive operation F^A of A, and each constant c^A of A, the functions defined by

$$f(\vec{x}) = F^A(\vec{x}) \qquad f(\vec{x}) = c^A$$

are in IND(A) and are of type $(n; \vec{k}, l)$.

II. *Projections.* The function f defined by

$$f(\vec{x}) = x_i$$

is in IND(A) and is of type $(n; \vec{k}, k_i)$.

III. *Definition by cases.* The function f defined by

$$f(b,x,y) = \begin{cases} x & \text{if } b = \mathbb{t} \\ y & \text{if } b = \mathbb{f} \end{cases}$$

is in IND(A) and is of type $(3; \mathsf{B}, k, k, k)$.

OPERATIONS

IV. *Composition.* Let $g_1, \ldots, g_m \in \mathrm{IND}(A)$ be functions with g_i of type $(n; \vec{k}, l_i)$ for $i = 1, \ldots, m$ and let $h \in \mathrm{IND}(A)$ be a function of type $(m; \vec{l}, l)$. Then f defined by

$$f(\vec{x}) \simeq h(g_1(\vec{x}), \ldots, g_m(\vec{x}))$$

is a function in IND(A) of type $(n; \vec{k}, l)$.

V. *Simultaneous primitive recursion on* $\mathbb{N}$. Let $g_1, \ldots, g_m \in \mathrm{IND}(A)$ be functions with g_i of type $(n; \vec{k}, l_i)$ for $i = 1, \ldots, m$, and let $h_1, \ldots, h_m \in \mathrm{IND}(A)$ be functions with h_i of type $(1+n+m; \mathsf{N}, \vec{k}, \vec{l}, l_i)$ for $i = 1, \ldots, m$. Then $f_1, \ldots, f_m$ defined by

$$\begin{aligned} f_1(0,\vec{x}) &\simeq g_1(\vec{x}) \\ &\vdots \\ f_m(0,\vec{x}) &\simeq g_m(\vec{x}) \\ f_1(z+1,\vec{x}) &\simeq h_1(z,\vec{x},f_1(z,\vec{x}), \ldots, f_m(z,\vec{x})) \\ &\vdots \\ f_m(z+1,\vec{x}) &\simeq h_m(z,\vec{x},f_1(z,\vec{x}), \ldots, f_m(z,\vec{x})) \end{aligned}$$

are functions in IND(A) with f_i of type $(n+1, \mathsf{N}, \vec{k}, l_i)$.

VI. *Least number operator on* $\mathbb{N}$. *Let* $g \in \mathrm{IND}(A)$ be a function of type $(n+1; \vec{k}, \mathsf{N}, \mathsf{B})$. Then f defined by

$$f(\vec{x}) \simeq \mu z[g(\vec{x}, z) = \mathrm{tt}]$$

is a function in $\mathrm{IND}(A)$ of type $(n; \vec{k}, \mathsf{N})$. Here $f(\vec{x}) \downarrow z$ if, and only if, $g(\vec{x}, y) \downarrow \mathrm{ff}$ for each $y < z$ and $g(\vec{x}, z) \downarrow \mathrm{tt}$.

4.1.2 Informal definition of computability on A

The set $\mathrm{IND}(A)$ of all partial functions of the form $f: A[\vec{k}] \to A_l$, obtained from the basic functions I—III by means of the operations IV—VI, is the set of partial functions *inductively definable* or *inductively computable* on A.

We have chosen the name *inductively* definable to emphasize the important role of the induction principle on the natural numbers $\mathbb{N}$ in the abstract computational formalism; and to maintain a simple terminology, suited to the tasks required of the formalism in this book.

The clauses for primitive recursion and the least number operator on $\mathbb{N}$ are mathematically the most significant. The set $\mathrm{PIND}(A)$ of functions obtained from clauses I—V we will term the set of *primitive inductively definable functions*, since those functions generalize the (simultaneous) primitive recursive functions on $\mathbb{N}$. Notice that the members of $\mathrm{PIND}(A)$ are all *total functions* and, therefore, it is clause VI that introduces the partial functions in $\mathrm{IND}(A)$.

Our adoption of a *simultaneous* primitive recursion clause in place of the (expected) single function scheme,

$$f(0, \vec{x}) \simeq g(\vec{x})$$
$$f(z+1, \vec{x}) \simeq h(z, \vec{x}, f(z, \vec{x})),$$

is also noteworthy.

In the single-sorted case of the natural numbers $\mathbb{N}$, primitive recursion and simultaneous recursion are equivalent methods of definition — thanks to the primitive recursive pairing functions on $\mathbb{N}$: see Kleene [1952] §46, or Peter [1967] p. 62.

In the many-sorted algebraic situation, however, simultaneous primitive recursion allows an important kind of interdependency of the different domains in the definition of functions. This interdependency is well exemplified in the construction of various syntactic term evaluation mappings (already seen in 3.6.4, and to be seen again on several occasions in this chapter). Thus, primitive recursion and simultaneous primitive recursion are not equivalent in the many-sorted situation; the appropriateness

and mathematical attractions of using simultaneous primitive recursion will be self-evident in due course, we hope.

4.1.3 Vector mappings

Consider a partial (vector-valued) mapping $f: A[\vec{k}] \to A[\vec{l}]$ of type $(n, m; \vec{k}, \vec{l})$ with coordinate partial functions $f_i: A[\vec{k}] \to A_{l_i}$ for $i = 1, \ldots, m$. Thus, for $\vec{x} \in A[\vec{k}]$

$$f(\vec{x}) \simeq (f_1(\vec{x}), \ldots, f_m(\vec{x}))$$

and (recall Section 2.1)

$$f(\vec{x})\downarrow \text{ iff } \text{for } i = 1, \ldots, m, \ f_i(x)\downarrow$$

and

$$f(\vec{x})\uparrow \text{ iff } \text{for } i = 1, \ldots, m, \ f_i(x)\uparrow.$$

Then the map f is said to be *inductively definable* or *inductively computable* on A if each of its coordinate functions f_i is in IND(A).

Notice that the simultaneous primitive recursion clause can be used to specify the coordinate mappings of a vector map.

4.1.4 Criticism

Mathematically, this definition 4.1.2 is not quite complete because a justification is needed for operation V; such an argument is easy to supply (see 4.1.6). Then, with *conventional rigour*, we have a *conventional semantical* definition of the inductively computable functions on A. Strictly speaking, this definition is not rigorous without a syntactic description of the generating process. For example, without such a syntactic component to the definition, the nature of arguments based on induction on the complexity of function definitions is not clear. The distinction between syntax and semantics is usually ignored in definitions of the partial recursive functions on $\mathbb{N}$ and could be safely ignored in this modest generalization to A, because of the familiar role of $\mathbb{N}$ in the algebra A. (To ignore the distinction is not consistent with the objectives of our field of study, of course!)

However, we are *not* interested in computation on a single structure A, but in processes which are *uniformly computable* over a class of structures $\mathbb{K}$. Indeed, for our purposes, computation on a single structure A *is* computation over the isomorphism type of A. Thus, we have need of a syntactic description of the inductively definable functions with a semantics of the form

$$\mathrm{IND}(\mathbb{K}) = \{\mathrm{IND}(A) \mid A \in \mathbb{K}\}.$$

This syntax we call *induction schemes.*

4.1.5 Induction Schemes

An induction scheme is a notation for a family of inductively definable functions which specifies a common structure for the functions. We will choose a particularly compact and abstract notation as our induction schemes suited to the mathematical work at hand. Let Σ be a signature with distinguished numerical and boolean sorts N and B (as in 1.1.1). The induction schemes are defined inductively, using items from Σ, natural numbers and the letters O,P,D,C,R,L to distinguish the six cases corresponding with the six clauses of 4.1.1.

I. For each function symbol F of Σ of type $(n; \vec{k}, l)$ with $n > 0$, the notation

$$< \mathrm{O}, F, n, \vec{k}, l >$$

is an induction scheme of type $(n; \vec{k}, l)$. And for each constant symbol c in Σ of type $(0; l)$ and any list of sorts $\vec{k}$, the notation

$$< \mathrm{O}, c, l, \vec{k} >$$

is an induction scheme of type $(n; \vec{k}, l)$.

II. For any list of sorts $\vec{k}$ and any $1 \leq i \leq n$, the notation

$$< \mathrm{P}, n, \vec{k}, i >$$

is an induction scheme of type $(n; \vec{k}, k_i)$.

III. For any sort k, the notation

$$< \mathrm{D}, k >$$

is an induction scheme of type $(3; \mathsf{B}, k, k, k)$.

IV. For any induction schemes $\alpha_1, \ldots, \alpha_m$ with α_i of type $(n; \vec{k}, l_i)$ and any induction scheme β of type $(m; \vec{l}, l)$, the notation

$$< \mathrm{C}, n, m, \alpha_1, \ldots, \alpha_m, \beta >$$

is an induction scheme of type $(n; \vec{k}, l)$.

V. For any induction schemes $\alpha_1, \ldots, \alpha_m$ with α_i of type $(n; \vec{k}, l_i)$ and any induction schemes $\beta_1, \ldots, \beta_m$ with β_i of type $(1+n+m; \mathsf{N}, \vec{k}, \vec{l}, l_i)$, the notation

$$<\mathrm{R}, n, m, i, \alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_m>$$

is an induction scheme of type $(n+1; \mathsf{N}, \vec{k}, l_i)$.

VI. For any induction scheme α of type $(n+1; \vec{k}, \mathsf{N}, \mathsf{B})$, the notation

$$<\mathrm{L}, n, \alpha>$$

is an induction scheme of type $(n; \vec{k}, \mathsf{N})$.

The set of induction schemes defined by clauses I—VI we denote $\boldsymbol{IndSch} = \boldsymbol{IndSch}(\Sigma)$. We use the notation $\alpha, \beta, \ldots$ to denote schemes.

EXERCISE. Devise an alternate concrete syntax to complement the abstract syntax of induction schemes; that is, by means of syntactic 'sugar', devise a pleasant functional programming notation whose semantics is intended to be the inductively definable mappings.

4.1.6 Semantics of Schemes

The meaning of a scheme α of type $(m; \vec{k}, l)$ on the class $\mathbb{K}$ is a family of mappings

$$\mathcal{M}_A(\alpha): A[\vec{k}] \to A_l$$

indexed by the structures $A \in \mathbb{K}$; that is, the semantics of α is of the form

$$\mathcal{M}_{\mathbb{K}}(\alpha) = \{\mathcal{M}_A(\alpha) \mid A \in \mathbb{K}\}.$$

The semantics of schemes on A is informally described in 4.1.1, of course; but formally, $\mathcal{M}_A(\alpha)$ must be defined by induction on the construction of α. Let us outline this definition.

The basic cases I—III are quite simple. For instance, in case II, if $\alpha \equiv <\mathrm{P}, m, \vec{k}, i>$ then $\mathcal{M}_A(\alpha)$ is the projection function $A[\vec{k}] \to A_{k_i}$,

$$\mathcal{M}_A(\alpha)(\vec{x}) \simeq x_i$$

as defined in case II of 4.1.1. Consider the induction steps. Composition is routine: suppose

$$\alpha \equiv <\mathrm{C}, n, m, \alpha_1, \ldots, \alpha_m, \beta>$$

and for $i = 1, \ldots, m$

$$\mathcal{M}_A(\alpha_i): A[\vec{k}] \to A_{l_i} \quad \text{and} \quad \mathcal{M}_A(\beta): A[\vec{l}] \to A_l.$$

Then we define $\mathcal{M}_A(\alpha): A[\vec{k}] \to A_l$ by

$$\mathcal{M}_A(\alpha)(\vec{x}) \simeq \mathcal{M}_A(\beta)(\mathcal{M}_A(\alpha_1)(\vec{x}), \ldots, \mathcal{M}_A(\alpha_m)(\vec{x}))$$

for all $\vec{x} \in A[\vec{k}]$. (Remember the standard convention for the composition

of partial functions in Section 2.1.)

In the case of primitive recursion, to define $\mathcal{M}_A(\alpha)$ for

$$\alpha \equiv <\mathrm{R}, n, m, i, \alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_m>$$

we need the following fact about the solutions to equations of the form V in 4.1.1:

LEMMA. *Let* $A \in \mathbb{K}$. *Let* $g_1, \ldots, g_m$ *be partial functions on* A *with* g_i *of type* $(n; \vec{k}, l_i)$ *and let* $h_1, \ldots, h_m$ *be partial functions on* A *with* h_i *of type* $(1+n+m; \mathsf{N}, \vec{k}, \vec{l}, l_i)$. *Then there exists a* unique *m-tuple of partial functions* $\vec{f} = (f_1, \ldots, f_m)$ *on* A *with* f_i *of type* $(1+n; \mathsf{N}, \vec{k}, l_i)$ *such that*

$$f_i(0, \vec{x}) \simeq g_i(\vec{x})$$

$$f_i(z+1, \vec{x}) \simeq h_i(z, \vec{x}, f_1(z, \vec{x}), \ldots, f_m(z, \vec{x})).$$

Furthermore, if $f_i(z, \vec{x})\uparrow$ *for any* i *and any* $z \in \mathbb{N}$ *then* $f_i(y, \vec{x})\uparrow$ *for* all i *and* all $y > z$.

Given the lemma, we can define $\mathcal{M}_A(\alpha) = f_i$ when for $j = 1, \ldots, m$,

$$\mathcal{M}_A(\alpha_j) = g_i \quad \text{and} \quad \mathcal{M}_A(\beta_j) = h_i.$$

PROOF OF LEMMA. The uniqueness of the solution to the equation is straight-forward: given solutions $\vec{f} = (f_1, \ldots, f_m)$ and $\vec{f}' = (f_1', \ldots, f_m')$ one proves that for each i

$$f_i(z, \vec{x}) \simeq f_i'(z, \vec{x})$$

by simultaneous induction on z.

To prove the existence of a solution, a map $\vec{f} = (f_1, \ldots, f_m)$ is constructed from a (unique) sequence $\{(f_1^z, \ldots, f_m^z) \mid z \in \mathbb{N}\}$ of approximating partial functions which satisfy the equations on any argument $(y, \vec{x})$ with $y < z$, and are undefined for all $(y, \vec{x})$ with $y \geq z$. These f_i^z are defined inductively as follows:

Basis $z = 0$. Let $f_i^0(y, \vec{x})\uparrow$, *i.e.* f_i^0 is the everywhere undefined function.

Basis $z = 1$. Let

$$f_i^1(y, \vec{x}) \simeq \begin{cases} f_i^0(y, \vec{x}) & \text{if } y \neq 0 \\ g_i(\vec{x}) & \text{if } y = 0 \end{cases}$$

Induction step. Let

$$f_i^{z+1}(y,\vec{x}) \simeq \begin{cases} f_i^z(y,\vec{x}) & \text{if } y \neq z \\ h_i(\vec{x},z,f_1^z(z,\vec{x}),\ldots,f_m^z(z,\vec{x})) & \text{if } y = z \end{cases}$$

One can now prove that the functions satisfy the requirements, by induction on z; and then prove that $\vec{f}=(f_1,\ldots,f_m)$ with

$$f_i(z,\vec{x}) \simeq f_i^{z+1}(z,\vec{x})$$

is the solution. □

Finally, in the case of the least number search operator on $\mathbb{N}$, if

$$\alpha \equiv <\mathrm{L}, n, \beta>$$

then to define $\mathcal{M}_A(\alpha)$ we can simply write

$$\mathcal{M}_A(\alpha)(\vec{x}) \simeq \begin{cases} z & \text{if } \mathcal{M}_A(\beta)(\vec{x},z)\downarrow\Uparrow \\ & \quad \text{and, for each } y<z,\ \mathcal{M}_A(\beta)(\vec{x},y)\downarrow\Uparrow \\ \uparrow & \text{if there is no such } z\,. \end{cases}$$

This concludes the definition of the semantics of schemes on a structure A.

We will usually write $\{\alpha\}^A$ for the map $\mathcal{M}_A(\alpha)$.

The semantics of α on the class $\mathbb{K}$ can now be formally defined:

$$\begin{aligned} \mathcal{M}_{\mathbb{K}}(\alpha) &= \{\mathcal{M}_A(\alpha) \mid A\in\mathbb{K}\} \\ &= \{\{\alpha\}^A \mid A\in\mathbb{K}\} \end{aligned}$$

And we can frame one of the basic definitions of the chapter.

4.1.7 Formal definition of computability over $\mathbb{K}$

A family $f=\{f_A \mid A\in\mathbb{K}\}$ of partial functions of type $(n;\vec{k},l)$ is said to be *inductively definable uniformly over* $\mathbb{K}$ if there exists an induction scheme α of type $(n;\vec{k},l)$ such that for every $A\in\mathbb{K}$ and every $\vec{x}\in A[\vec{k}]$,

$$f_A(\vec{x}) \simeq \{\alpha\}^A(\vec{x}).$$

4.1.8 Vector mappings

A family $f=\{f_A \mid A\in\mathbb{K}\}$ of (vector-valued) partial mappings of type $(n,m;\vec{k},\vec{l})$ is said to be *inductively definable uniformly over* $\mathbb{K}$ if there

exists a list of induction schemes $\vec{\alpha}=(\alpha_1,\ldots,\alpha_m)$, with α_i of type $(n;\vec{k},l_i)$, which uniformly define the families $f_i=\{f_A^i \mid A\in\mathbb{K}\}$ of coordinate functions of f (recall 4.1.3).

We will usually write $\{\vec{\alpha}\}^A$ as an abbreviation for the list of maps $\{\alpha_1\}^A,\ldots,\{\alpha_m\}^A$; in particular,

$$\{\vec{\alpha}\}^A(\vec{x}) \simeq (\{\alpha_1\}^A(\vec{x}),\ldots,\{\alpha_m\}^A(\vec{x})).$$

4.1.9 Functions on $\mathbb{K}^u$ and $\mathbb{K}^*$

The sole purpose of the structures A^u, and A^*, is the semantical interpretation of programs, and assertions about the behaviour of programs, that compute on the structure $A\in\mathbb{K}$. However, we have technical need of a special theory of inductively definable functions on both $\mathbb{K}^u$ and $\mathbb{K}$, in order to conclude the logical definability work in Chapter 3 (see 3.6.4), and in connection with a theorem in Section 4.7 (Basic Lemma 4.7.3). Moreover, it is interesting to note that inductive definability on $\mathbb{K}^u$ can be simulated by inductive definability on $\mathbb{K}$.

First we observe that the structures A^u and A^* for $A\in\mathbb{K}$ are not, strictly speaking, standard structures because their numerical and boolean domains are $\mathbb{N}^u$ and $\mathbb{B}^u$. From the point of view of our definitions, the addition of u can be controlled by the unary predicate

$$\boldsymbol{Unspec}_i(x) = \begin{cases} \mathbb{t} & \text{if } x = u \\ \mathbb{f} & \text{if } x \neq u \end{cases}$$

for $i=\mathsf{N},\mathsf{B}$. Thus, we now assume that A^u contains these predicates. The structure A^*, as defined in 3.2.1, already contains these predicates.

4.1.10 Case of $\mathbb{K}^u$

To define the class of inductively definable functions on A^u we must adapt the clauses in 4.1.1 involving the domains $\mathbb{N}$ and $\mathbb{B}$ of A. The first such clause is that of definition by cases III, which is replaced by

$$f(b,x,y) = \begin{cases} x & \text{if } \boldsymbol{Unspec}_{\mathsf{B}}(b)=\mathbb{f} \text{ and } b=\mathbb{t} \\ y & \text{if } \boldsymbol{Unspec}_{\mathsf{B}}(b)=\mathbb{f} \text{ and } b=\mathbb{f} \\ u & \text{if } \boldsymbol{Unspec}_{\mathsf{B}}(b)=\mathbb{t}. \end{cases}$$

Next, we must add to clause V, about primitive recursion, the new basis case

$$f_i(\mathbb{u}_{\mathbb{N}}, \vec{x}) \simeq \mathbb{u}_{l_i}$$

where $i = 1, \ldots, m$. And, finally, we must add to clause VI the case

$f(\vec{x})\downarrow \mathbb{u}$ iff for some $z \in \mathbb{N}$, $g(\vec{x}, y)\downarrow \mathbb{f}$ for each $y < z$, and $g(\vec{x}, z)\downarrow \mathbb{u}$.

4.1.11 Reduction of $\mathbb{K}^{\mathbb{u}}$ to $\mathbb{K}$

We construct a representation of $A^{\mathbb{u}}[\vec{k}]$ in terms of $A[\vec{k}]$ as follows: for each sort k, let $\alpha_k : \mathbb{B} \times A_k \longrightarrow A_k^{\mathbb{u}}$ be defined by

$$\alpha_k(b, x) = \begin{cases} x & \text{if } b = \mathbb{t} \\ \mathbb{u}_k & \text{if } b = \mathbb{f} \end{cases}$$

Now we set

$$A_{\mathbb{B}}[\vec{k}] = (\mathbb{B} \times A_{k_1}) \times \cdots \times (\mathbb{B} \times A_{k_n})$$

and define a representation by means of the surjection

$$\alpha[\vec{k}] : A_{\mathbb{B}}[\vec{k}] \longrightarrow A[\vec{k}],$$

$$\alpha[\vec{k}](\ldots, (b_i, x_i), \ldots) = (\ldots, \alpha_{k_i}(b_i, x_i), \ldots).$$

With this machinery we can effect the following reduction of inductive definability over $\mathbb{K}^{\mathbb{u}}$ to inductive definability over $\mathbb{K}$.

THEOREM. *Let $f = \{f_{A^{\mathbb{u}}} \mid A^{\mathbb{u}} \in \mathbb{K}^{\mathbb{u}}\}$ be a family of partial mappings that is inductively definable uniformly over $\mathbb{K}^{\mathbb{u}}$. Then there exists a family $\hat{f} = \{\hat{f}_A \mid A \in \mathbb{K}\}$ of partial mappings that is inductively definable uniformly over $\mathbb{K}$ such that for each $A \in \mathbb{K}$ the following diagram commutes:*

$$\begin{array}{ccc} A^{\mathbb{u}}[\vec{k}] & \xrightarrow{f_{A^{\mathbb{u}}}} & A^{\mathbb{u}}[\vec{l}] \\ \alpha[\vec{k}] \uparrow & & \uparrow \alpha[\vec{l}] \\ A_{\mathbb{B}}[\vec{k}] & \xrightarrow{\hat{f}_A} & A_{\mathbb{B}}[\vec{l}] \end{array}$$

PROOF. Induction on the complexity of the induction scheme for f. □

4.1.12 Case of $\mathbb{K}^*$

We leave it to the reader to verify that the solution adopted in 4.1.10 for inductive definability in $\mathbb{K}^u$ applies directly to $\mathbb{K}^*$. Let us note, however, that inductive definability on $\mathbb{K}^*$ *cannot* be reduced to inductive definability on $\mathbb{K}$ (Section 4.8).

4.2 SOME IMPORTANT PROPERTIES

We will gather a number of simple, but valuable, results about the inductively definable functions on $\mathbb{K}$.

4.2.1 The functions computable on $\mathbb{N}$

Consider the functions on the standard numerical domain $\mathbb{N}$ of a structure $A \in \mathbb{K}$ that are definable by the induction schemes; not surprisingly:

THEOREM. *Let A be a structure and f a partial function on A of type $(n; \mathbf{N}, \ldots, \mathbf{N}, \mathbf{N})$, i.e. a function $f: \mathbb{N}^n \to \mathbb{N}$. If f is partial recursive on $\mathbb{N}$ then f is definable by an induction scheme over A. If f is primitive recursive then f is definable by a primitive induction scheme over A.*

PROOF. Obvious, on consulting Kleene [1952], §43 and §63. □

A similar result can be framed and proved for *r.e.*, *recursive* and *primitive recursive* relations on $\mathbb{N}$, involving the boolean domain $\mathbb{B}$. But the converse of the theorem is false, since a structure may be chosen to have non-recursive basic operations.

4.2.2 The manipulation of arguments

The elementary and essential operations of *permuting the arguments* of a function, *identifying the arguments* of a function, and introducing *dummy arguments* etc., to make new functions from old ones, are all applicable to IND($\mathbb{K}$). These operations can be stated more precisely and uniformly as follows:

PROPOSITION *Let $(n; \vec{k}, l)$ be a type. Consider a total mapping $\pi: \{1, \ldots, m\} \to \{1, \ldots, n\}$ and set $\pi(\vec{k}) = (k_{\pi(1)}, \ldots, k_{\pi(m)})$. Given an induction scheme α of type $(m; \pi(\vec{k}), l)$, there exists an induction scheme β of type $(n; \vec{k}, l)$ such that for all $A \in \mathbb{K}$, $\vec{x} \in A[\vec{k}]$*

$$\{\beta\}^A(\vec{x}) \simeq \{\alpha\}^A(x_{\pi(1)}, \ldots, x_{\pi(m)}).$$

In the special case of *permuting arguments*, $m = n$ and π is bijective. In one of the special cases of *identifying arguments*, $m > n$ and π is

surjective. In the special case of adding *dummy arguments*, $m < n$ and π is the identity function on $\{1, \ldots, m\}$. Such function building techniques are based on the projection and composition of schemes and are discussed in Kleene [1952], §44.

4.2.3 General definition by cases

The following result is a very general form of definition by cases.

PROPOSITION. *Let $\alpha_1, \ldots, \alpha_m$ by induction schemes of type $(n; \vec{k}, l)$ and let $\beta_1, \ldots, \beta_m$ be induction schemes of type $(n; \vec{k}, \mathsf{B})$. Then there is an induction scheme γ of type $(n; \vec{k}, l)$ such that for every $A \in \mathbb{K}$, $\vec{x} \in A[\vec{k}]$*

$$\{\gamma\}^A(\vec{x}) \simeq \begin{cases} \{\alpha_1\}^A(\vec{x}) & \text{if } \{\beta_1\}^A(\vec{x}) \downarrow \Uparrow \\ \{\alpha_2\}^A(\vec{x}) & \text{if } \{\beta_1\}^A(\vec{x}) \downarrow \Uparrow \text{ and } \{\beta_2\}^A(\vec{x}) \downarrow \Uparrow \\ \quad \vdots & \\ \{\alpha_m\}^A(\vec{x}) & \text{if } \{\beta_i\}^A(\vec{x}) \downarrow \Uparrow \text{ for } 1 \le i < m \\ & \quad \text{and } \{\beta_m\}^A(\vec{x}) \downarrow \Uparrow . \end{cases}$$

Notice that $\{\gamma\}^A(\vec{x}) \uparrow$ if, and only if, for some $1 \le i \le m$,

$$\{\alpha_j\}^A(\vec{x}) \downarrow \Uparrow \text{ for } j \le i \text{ and } \{\alpha_i\}^A(\vec{x}) \uparrow .$$

4.2.4 Iteration lemma

One of the important properties of the primitive recursive functions on $\mathbb{N}$ is this: for any primitive recursion function $f : \mathbb{N} \to \mathbb{N}$, the iteration function $I(f) : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by

$$\begin{aligned} I(f)(n, x) &= (f \circ \cdots \circ f)(x) \qquad (n \text{ times}) \\ &= f^n(x) \end{aligned}$$

is again primitive recursive. Here is a generalization of this property which is useful in unfolding looping constructs such as the **'while'** statement.

ITERATION LEMMA. *Let $f = \{f_A \mid A \in \mathbb{K}\}$ be a family of partial mappings of type $(n, n; \vec{k}, \vec{k})$ that is uniformly definable over $\mathbb{K}$ by induction schemes $\vec{\alpha}$. Then the family $I = \{I_A \mid A \in \mathbb{K}\}$ of iteration maps*

$$I_A : \mathbb{N} \times A[\vec{k}] \to A[\vec{k}]$$

defined by

$$I_A(z,\vec{x}) = (f_A \circ \cdots \circ f_A)(\vec{x}) \qquad (z \text{ times})$$
$$= f_A^z(\vec{x})$$

is uniformly definable over $\mathbb{K}$ *by induction schemes* $\vec{\beta}$.

PROOF. Suppose $f = \{f_A \mid A \in \mathbb{K}\}$ is defined by $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$, where α_i defines the family $f_i = \{f_A^i \mid A \in \mathbb{K}\}$ of i-th coordinate maps of f. We consider the construction of the iteration map I_A for f_A on some arbitrary structure $A \in \mathbb{K}$.

Now each coordinate map I_A^i of I_A can be defined, informally, by the following primitive recursion:

$$I_A^i(0,\vec{x}) \simeq x_i$$
$$I_A^i(z+1,\vec{x}) \simeq f_A^i(I_A^1(z,\vec{x}), \ldots, I_A^n(z,\vec{x})).$$

This observation suggests that an obvious n-tuple of schemes $\vec{\beta} = (\beta_1, \ldots, \beta_n)$ to define the family $I = \{I_A \mid A \in \mathbb{K}\}$, uniformly over $\mathbb{K}$, is

$$\beta_i \equiv \langle \mathrm{R}, n, n, i, \gamma_1, \ldots, \gamma_n, \alpha_1, \ldots, \alpha_n \rangle$$

wherein γ_i is a scheme that computes the i-th projection function of type $(n; \vec{k}, k_i)$ over $\mathbb{K}$.

We leave to the interested reader the task of proving that for each i, $A \in \mathbb{K}$ and $\vec{x} \in A[\vec{k}]$,

$$\{\beta_i\}^A(z,\vec{x}) \simeq I_A^i(z,\vec{x})$$

by induction on z. □

4.3 FROM INDUCTION SCHEMES TO 'WHILE' PROGRAMS

A program S specifies or "computes" a family of state transformations,

$$\mathcal{M}_A(S): \mathrm{PR.STATE}(A) \dashrightarrow \mathrm{STATE}(A)$$

indexed by $A \in \mathbb{K}$, which records the results of executing S on all possible values of its program variables. Among the effects included is the *unspecified value* $\mathbb{u}$ of a variable which may, or may not, lead to an *error state* ε in a computation. Notice, however, that these program-theoretic items have nothing to do with the inductively definable functions over $\mathbb{K}$ and the mathematical mechanisms that calculate them. We begin the comparison of induction schemes and programs by specifying some simple conditions under which a program computes a function on a structure A.

4.3.1 Functions computable by programs

Let S be any program of one of the kinds considered in the previous chapters. Let $\vec{v} \equiv (v_1, \ldots, v_n)$ and $\vec{w} \equiv (w_1, \ldots, w_m)$ be lists of distinct simple variables of sorts $\vec{k} = (k_1, \ldots, k_n)$ and $\vec{l} = (l_1, \ldots, l_m)$ respectively. The triple $[S, \vec{v}, \vec{w}]$ will be called a *program with input variables* $\vec{v}$ *and output variables* $\vec{w}$ or simply an *i/o-program.*

DEFINITION. A family $f = \{f_A \mid A \in \mathbb{K}\}$ of partial mappings of type $(n, m; \vec{k}, \vec{l})$ is said to be *computable by an i/o-program* $[S, \vec{v}, \vec{w}]$ if for every $A \in \mathbb{K}$ and $\vec{x} \in A[\vec{k}]$, and for every $\sigma \in \mathrm{PR.STATE}(A)$ with $\sigma(v_i) = x_i$ $(1 \leq i \leq n)$,

$$f_A(\vec{x}) \downarrow \vec{y} \quad \text{iff} \quad \mathcal{M}_A(S)(\sigma) \downarrow \sigma' \neq \varepsilon$$
$$\text{where } \sigma'(w_i) = y_i \neq \mathbb{u} \text{ for all } i = 1, \ldots, m; \text{ and}$$
$$f_A(\vec{x}) \uparrow \quad \text{iff} \quad \mathcal{M}_A(S)(\sigma) \uparrow.$$

Notice that we require a program S to compute the family f in such a way that *divergence of* f *corresponds* only *with non-termination of* S *and* not *with the occurrence of errors or unspecified values for output variables.*

4.3.2 Functional programs

The last definition leads to the interesting semantical concept of a $\mathbb{K}$-*functional program* as that of an i/o-program $[S, \vec{v}, \vec{w}]$ in which the values of the output variables $\vec{w}$ are uniquely determined by the values of the input variables $\vec{v}$.

DEFINITION An i/o program $[S, \vec{v}, \vec{w}]$ is called a $\mathbb{K}$-*functional program* if given any $A \in \mathbb{K}$ and any $\sigma_1, \sigma_2 \in \mathrm{PR.STATE}(A)$ such that $\sigma_1(v_i) = \sigma_2(v_i) \neq \mathbb{u}$ for $1 \leq i \leq n$,

$$\text{either} \quad \mathcal{M}_A(S)(\sigma_1) \downarrow \sigma_1' \neq \varepsilon \text{ and } \mathcal{M}_A(S)(\sigma_2) \downarrow \sigma_2' \neq \varepsilon$$
$$\text{and } \sigma_1'(w_i) = \sigma_2'(w_i) \neq \mathbb{u} \text{ for } 1 \leq i \leq m$$
$$\text{or} \quad \mathcal{M}_A(S)(\sigma_1) \uparrow \text{ and } \mathcal{M}_A(S)(\sigma_2) \uparrow.$$

Notice that a $\mathbb{K}$-functional program never returns an error state provided its input variables are specified.

The property of functionality is akin to the properties seen in 1.2.13, 2.2.8 and 3.1.11: provided σ_1, σ_2 are specified on $\vec{v}$ we have

$$\sigma_1 \simeq \sigma_2 \ (\mathrm{rel}\ \vec{v}) \implies \mathcal{M}_A(S)(\sigma_1) \simeq \mathcal{M}_A(S)(\sigma_2) \ (\mathrm{rel}\ \vec{w}).$$

In a natural way, such a $\mathbb{K}$-functional program $[S, \vec{v}, \vec{w}]$ computes a

function f_A of type $(n,m;\vec{k},\vec{l})$ on any $A\in\mathbb{K}$ by defininining for each $\vec{x}\in A[\vec{k}]$ the special state $[\vec{x}]\in$PR.STATE(A) by

$$[\vec{x}](v)=\begin{cases} x_i & \text{if } v=v_i \text{ for } i=1,\ldots,n \\ \text{ш} & \text{otherwise} \end{cases}$$

and then setting

$$f_A(\vec{x})=\vec{y} \quad \text{if } \mathcal{M}_A(S)([\vec{x}])\downarrow\sigma\neq\varepsilon \text{ such that } \sigma(w_i)=y_i \text{ for } i=1,\ldots,m;$$

$$f_A(\vec{x})\uparrow \quad \text{if } \mathcal{M}_A(S)([\vec{x}])\uparrow.$$

Since $[S,\vec{v},\vec{w}]$ is a functional program the choice of $[\vec{x}]$ is not necessary, of course: for any σ

$$\sigma\simeq[\vec{x}]\ (\text{rel }\vec{v}) \implies \mathcal{M}_A(S)(\sigma)\simeq\mathcal{M}_A(S)([\vec{x}])\ (\text{rel }\vec{w});$$

and hence choosing equivalent σ (rel $\vec{v}$) would define the same function $f_A\colon A[\vec{k}]\to A[\vec{l}]$.

Thus, to any $\mathbb{K}$-functional program $[S,\vec{v},\vec{w}]$ we may associate a functional meaning or semantics

$$\boldsymbol{fn}(S,\vec{v},\vec{w})=\{\boldsymbol{fn}_A(S,\vec{v},\vec{w})\mid A\in\mathbb{K}\}$$

by taking $\boldsymbol{fn}_A(S,\vec{v},\vec{w})$ to be the function $f_A\colon A[\vec{k}]\to A[\vec{l}]$ described above.

The functional behaviour of 'while' and 'while'-array programs, and von Neumann programs in general, is an interesting and useful subject, both theoretically and practically. Starting from the fact that the set of i/o programs $[S,\vec{v},\vec{w}]$ that are functional on the natural numbers $\mathbb{N}$ is not a recursively enumerable set, Clive Jervis has devised syntactic conditions on programs to ensure functional behaviour. These sets of programs are polynomial-time decidable and implement all computable functions: see Jervis [1987]. Jervis' conditions have also been implemented in a compiler for a language CARESS (Martin and Tucker [1987]).

4.3.3 'while' computable functions

A family $f=\{f_A\mid A\in\mathbb{K}\}$ of partial functions of type $(n,m;\vec{k},\vec{l})$ is said to be *computable by a* 'while' *program uniformly over* $\mathbb{K}$ if there is an appropriate functional 'while' program $[S,\vec{v},\vec{w}]$ such that for all $A\in\mathbb{K}$ and all $\vec{x}\in A[\vec{k}]$,

$$f_A(\vec{x})\simeq\boldsymbol{fn}_A(S,\vec{v},\vec{w})(\vec{x}).$$

4.3.4 Scheme definability implies 'while' program computability

THEOREM *Let* $f=\{f_A \mid A\in\mathbb{K}\}$ *be a family of partial mappings of type* $(n,m;\vec{k},\vec{l})$ *that is definable by induction schemes* $\vec{\alpha}$ *uniformly over* $\mathbb{K}$. *Then* f *is computable by a functional* 'while' *program* $[S,\vec{v},\vec{w}]$ *uniformly over* $\mathbb{K}$. *Moreover, one can effectively calculate* $[S,\vec{v},\vec{w}]$ *from* $\vec{\alpha}$.

PROOF. We consider the coordinate maps of f and how to construct a functional 'while' program $[S_i,\vec{v},w_i]$ for each scheme α_i of type $(n;\vec{k},l_i)$. Then we show how these functional programs can be combined to make the required functional program $[S,\vec{v},\vec{w}]$ to compute f.

Let α be any induction scheme of type $(n;\vec{k},l)$. We will construct a functional 'while' program $[S,\vec{v},w]$ for α by induction on the definition of α.

The basis cases are straight-forward. For example, consider definition by cases where $\alpha\equiv<\mathrm{D},k>$. Take

$$S \equiv \textsf{if } v^{\mathsf{B}} \textsf{ then } w:=v_1 \textsf{ else } w:=v_2 \textsf{ fi}$$

where v^{B} is a Boolean variable and w,v_1,v_2 are variables of type k. To show that the functional program $[S,(v^{\mathsf{B}},v_1,v_2),w]$ computes $\{\alpha\}^A$ on each $A\in\mathbb{K}$ is trivial.

Consider the induction step. In the case of composition,

$$\alpha \equiv <\mathrm{C},n,m,\alpha_1,\ldots,\alpha_m,\beta>.$$

Suppose that there are functional programs

$$[S_1,\vec{v}_1,w_1],\ldots,[S_m,\vec{v}_m,w_m],[\hat{S},\vec{y},z]$$

to uniformly compute $\{\alpha_1\}^A,\ldots,\{\alpha_m\}^A,\{\beta\}^A$ over all $A\in\mathbb{K}$. Without loss of generality we can assume *these programs have no variables in common.* This means that for all $A\in\mathbb{K}$ and $\sigma\in\mathrm{PR.STATE}(A)$:

if $\mathcal{M}_A(S_i)(\sigma)\downarrow$ then $\mathcal{M}_A(S_i)(\sigma)\simeq\sigma$ (rel $\boldsymbol{Var}(S_1,\ldots,S_{i-1},S_{i+1},\ldots,S_m,\hat{S})$),

if $\mathcal{M}_A(\hat{S})(\sigma)\downarrow$ then $\mathcal{M}_A(\hat{S})(\sigma)\simeq\sigma$ (rel $\boldsymbol{Var}(S_1,\ldots,S_m)$).

Let $\vec{v}$ and w be new variables and consider the following program:

$$
\begin{aligned}
S \equiv\ & \vec{v}_1 := \vec{v};\ S_1; \\
& \vec{v}_2 := \vec{v};\ S_2; \\
& \quad \vdots \\
& \vec{v}_m := \vec{v};\ S_m; \\
& \vec{y} := (w_1, \ldots, w_m);\ \hat{S}; \\
& w := z\ .
\end{aligned}
$$

Here notation of the form $\vec{y} := \vec{w}$ or $\vec{y} := (w_1, \ldots, w_m)$ does not indicate multiple assignments, but the sequence of assignments

$$y_1 := w_1;\ \cdots\ ; y_m := w_m\ .$$

The functional program $[S, \vec{v}, w]$ computes α over $\mathbb{K}$. For, clearly,

$$[\vec{v}_i := \vec{v};\ S_i,\ \vec{v}, w_i] \quad \text{and} \quad [\vec{y} := (w_1, \ldots, w_m);\hat{S},\ \vec{w}, z]$$

compute α_i and β over $\mathbb{K}$. And S is made to initialise the distinct input variables of the S_i by the common values of $\vec{v}$, and collect their output values by $\vec{y}$ in order to initialize the input variables of $\hat{S}$. Thus, S matches the distinct input and output variables.

EXERCISE. Verify the fact that $[S, \vec{v}, w]$ computes α over $\mathbb{K}$.

In the case of primitive recursion, let

$$\alpha \equiv <\mathrm{R}, n, m, j, \alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_m>$$

where α_i is of type $(n; \vec{k}, l_i)$ and β_i is of type $(n+1+m; \vec{k}, \mathsf{N}, \vec{l}, l_i)$. By the induction hypothesis, there are functional programs

$$[S_i, \vec{v}_i, w_i] \quad \text{and} \quad [\hat{S}_i, (\vec{y}_i, d_i, \vec{z}_i), s_i]$$

which compute α_i and β_i over $\mathbb{K}$. Again we assume *these programs have no variables in common.* Consider the following program S with new variables c, d of type N, and $\vec{v}, \vec{r}$ of types $\vec{k}, \vec{l}$:

$$
\begin{array}{l}
S \equiv\ \vec{v}_1 := \vec{v}\,;\, S_1\,;\ \ r_1 := w_1\,; \\
\qquad\qquad \vdots \\
\quad\ \vec{v}_m := \vec{v}\,;\, S_m\,;\ \ r_m := w_m\,; \\
\quad\ c := 0\,; \\
\quad\ \textsf{while}\ c < d\ \ \textsf{do}\ \ \vec{y}_1 := \vec{v}\,;\ d_1 := c\,;\ \vec{z}_1 := (r_1, \ldots, r_m)\,; \\
\qquad\qquad\qquad \hat{S}_1\,; \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad \vec{y}_m := \vec{v}\,;\ d_m := c\,;\ \vec{z}_m := (r_1, \ldots, r_m)\,; \\
\qquad\qquad\qquad \hat{S}_m\,; \\
\qquad\qquad\qquad r_1 := s_1\,; \\
\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad r_m := s_m\,; \\
\qquad\qquad\qquad c := c+1 \\
\qquad\qquad\ \textsf{od}.
\end{array}
$$

EXERCISE. Verify that the functional program $[S, (\vec{v}, d), r_j]$ computes the induction scheme α over $\mathbb{K}$.

In the case of the least number operator,

$$\alpha \equiv \langle \mathrm{L}, n, \beta \rangle$$

where β is of type $(n+1, \vec{k}, \mathsf{N}, \mathsf{B})$. By the induction hypothesis, there is a functional program $[\hat{S}, (\vec{y}, d), b]$, where b is a boolean variable, that computes β over $\mathbb{K}$. Consider the program S with new variables $\vec{v}$ of type $\vec{k}$ and c of type N:

$$
\begin{array}{l}
S \equiv\ c := 0; \\
\qquad b := \textsf{false}; \\
\qquad \textsf{while not}(b)\ \ \textsf{do}\ \ \vec{y} := \vec{v}\,;\ d := c\,; \\
\qquad\qquad\qquad\qquad\ \ \hat{S}\,; \\
\qquad\qquad\qquad\qquad\ \ c := c+1 \\
\qquad\qquad\qquad\ \ \textsf{od}; \\
\qquad c := c-1\,.
\end{array}
$$

EXERCISE. Verify that the functional program $[S, \vec{v}, c]$ computes α over $\mathbb{K}$.

We have shown how to make a functional program $[S, \vec{v}, w]$ for any induction scheme α of type $(n; \vec{k}, l)$. It remains for us to show how this method can be applied to the induction schemes $\alpha_1, \ldots, \alpha_m$ for the coordinate

mappings of f and the resulting functional programs $[S_1, \vec{v}_1, w_1], \ldots, [S_m, \vec{v}_m, w_m]$ combined to make the single functional program $[S, \vec{v}, \vec{w}]$ that computes f.

Using this method, it is easy to construct functional programs

$$[S_1, \vec{v}, w_1], \ldots, [S_m, \vec{v}, w_m]$$

for the induction schemes $\alpha_1, \ldots, \alpha_m$ where the input variables $\vec{v}$ are the only variables common to the $S_1, \ldots, S_m$ and, furthermore,

$$\boldsymbol{Var}(S_i) \cap \boldsymbol{Var}(S_j) = \{\vec{v}\}$$

for $i \neq j$. Put $\vec{w} = (w_1, \ldots, w_m)$.

In order to preserve the values of the input variables $\vec{v}$, choose new variables $\vec{x} = (x_1, \ldots, x_m)$ (disjoint from $\vec{v}, \vec{w}$ and all variables in $S_1, \ldots, S_m$), and define $S_i' \equiv \vec{v} := \vec{x}; S_i$ and $S \equiv \vec{x} := \vec{v}; S_1'; \cdots; S_m'$. We claim:

PROPOSITION. *The functional program* $[S, \vec{v}, \vec{w}]$ *computes* f.

This will follow from the

LEMMA. *For any* $i = 1, \ldots, m$ *and any* $\sigma \in \mathrm{PR.STATE}(A)$

$$\mathfrak{M}_A(S)(\sigma) \simeq \mathfrak{M}_A(S_i)(\sigma) \quad (\mathrm{rel}\ w_i).$$

PROOF. For these programs, our convention on coordinate mappings for vector mappings ensures that

$$\text{either } \mathfrak{M}_A(S_i')(\sigma)\downarrow \text{ for all } i = 1, \ldots, m$$
$$\text{or } \quad \mathfrak{M}_A(S_i')(\sigma)\uparrow \text{ for all } i = 1, \ldots, m.$$

Thus, we know that

$$\mathfrak{M}_A(S)(\sigma)\uparrow \quad \text{iff} \quad \mathfrak{M}_A(S_i')(\sigma)\uparrow \text{ for } i = 1, \ldots, m$$

and we can restrict our attention to the case where $\mathfrak{M}_A(S)(\sigma)\downarrow$.

By virtue of the construction of the S_i' we know the following important fact: for $i = 1, \ldots, m$

$$\mathfrak{M}_A(S_i')(\sigma) \simeq \sigma \quad (\mathrm{rel}\ \vec{x}, w_1, \ldots, w_{i-1})$$

i.e. S_i preserves inputs and all earlier computed outputs (thanks to the disjoint variables condition); whence it is easy to see that for $i = 1, \ldots, m$

$$\mathfrak{M}_A(S_1'; \ldots; S_{i-1}')(\sigma) \simeq \sigma \quad (\mathrm{rel}\ \vec{x}), \tag{1}$$

$$\mathfrak{M}_A(S_{i+1}'; \ldots; S_m')(\sigma) \simeq \sigma \quad (\mathrm{rel}\ w_i). \tag{2}$$

Thus, applying $\mathcal{M}_A(S_i')$ to (1) we obtain

$$\mathcal{M}_A(S_1';\ldots;S_i')(\sigma) \simeq \mathcal{M}_A(S_i')(\sigma) \quad (\text{rel } w_i)$$

because $[S_i', \vec{v}, w_i]$ is a functional program. Now using (2) we deduce that

$$\mathcal{M}_A(S_1';\ldots;S_m')(\sigma) \simeq \mathcal{M}_A(S_i')(\sigma) \quad (\text{rel } w_i),$$

from which the lemma follows by applying $\mathcal{M}_A(\vec{x} := \vec{v})$ to each side. □

4.4 FROM 'WHILE' PROGRAMS TO INDUCTION SCHEMES

We will simulate functional **'while'** program computation on STATE(A) with inductively definable functions on A and prove the converse of Theorem 4.3.4. To accomplish these tasks we first pay more attention to the semantics of function programs and the roles of $\mathbb{u}$ and ε.

4.4.1 Semantics of functional programs

Recall from 4.3.2 that a $\mathbb{K}$-functional program $[S, \vec{v}, \vec{w}]$ has functional semantics

$$\boldsymbol{fn}(S, \vec{v}, \vec{w}) = \{\boldsymbol{fn}_A(S, \vec{v}, \vec{w}) \mid A \in \mathbb{K}\}$$

that is most simply described as follows: for each $A \in \mathbb{K}$,

$$\boldsymbol{fn}_A(S, \vec{v}, \vec{w}): \; A[\vec{k}] \to A[\vec{l}]$$

is computed by giving $\vec{v}$ the values $\vec{x} \in A[\vec{k}]$, executing S, and taking $\boldsymbol{fn}_A(S, \vec{v}, \vec{w})(\vec{x})$ to be the values $\vec{y} \in A[\vec{l}]$ of $\vec{w}$ if and when S terminates. The values of the other variables of S prior and subsequent to the execution of S are immaterial; and, indeed, our semantics allows a direct formulation of this by saying: define special state

$$[\vec{x}](v) = \begin{cases} x_i & \text{if } v \equiv v_i \\ \mathbb{u} & \text{otherwise,} \end{cases}$$

and if $\mathcal{M}_A(S)([\vec{x}]) \downarrow \sigma$ then take $y_i = \sigma(w_i)$.

But it is important to note that the definition of a functional program requires that the values of the remaining variables be immaterial in the stronger sense that S computes the same values $\vec{y}$ for $\vec{w}$ on the entire neighbourhood of states

$$\text{NBHD}(\vec{x}) =_{df} \{\sigma \in \text{PR.STATE}(A) \mid \sigma(v_i) = x_i \text{ for } i = 1, \ldots, n\}.$$

Thus, while the value $\vec{y} = \boldsymbol{fn}_A(S, \vec{v}, \vec{w})(\vec{x})$ is uniquely determined by $\vec{x}$,

each *computation* of $\vec{y}$ by program S depends on the choice of its initial state $\sigma \in \text{NBHD}(\vec{x})$. Because of the common values $\vec{y}$ of $\vec{w}$, we know that *not one of these computations by S produces the error state* ε:

OBSERVATION. *If* $[S, \vec{v}, \vec{w}]$ *is a* $\mathbb{K}$*-functional program then for every* $A \in \mathbb{K}$ *and* $\vec{x} \in A[\vec{k}]$, *we may restrict the state transformation to*

$$\mathfrak{M}_A(S)\colon \text{NBHD}(\vec{x}) \dashrightarrow \text{PR.STATE}(A).$$

In this section, for 'while' programs, and in Section 4.7, for 'while'-array programs, we will make representations of certain special computations by functional programs, in order to prove that their functional semantics is inductively definable. These sets of computations will be selected by choosing sets of initial states, on which to execute programs, and will depend on the type of programming language involved; the process will be referred to as *localizing* computation on the space of proper states. In both cases of 'while' and 'while'-array programs, we will consider computation by a program S localized to the set of all initial states which are specified on some super-set of the variables of S.

4.4.2 A standard representation of the state space and state transformation

We will represent the official state transformation

$$\mathfrak{M}_A(S)\colon \text{PR.STATE}(A) \dashrightarrow \text{STATE}(A)$$

of a 'while' program S as a partial mapping $T_A(S)$ on A. The method of construction conforms with the representation of states described and used earlier (in 2.6.5 and 3.6.4), and is in some respects simpler.

First, let S be an arbitrary 'while' program whose variables are among the simple variables $\vec{v} = (v_1, \ldots, v_M)$ of sorts $\vec{k} = (k_1, \ldots, k_M)$ respectively (compare 2.6.4 for this notation); in symbols $\boldsymbol{Var}(S) \subseteq \vec{v}$. Consider the effect of executing S on the states in

$$\text{SP.STATE}(\vec{v}, A) =_{df} \{\sigma \in \text{PR.STATE}(A) \mid \text{for } 1 \leq i \leq M,\ \sigma(v_i) \neq \text{Ш}\}$$

i.e. on proper states with all variables of S specified.

By Proposition 1 in 2.2.10, for $\sigma \in \text{SP.STATE}(\vec{v}, A)$,

either $\mathfrak{M}_A(S)(\sigma) \downarrow \sigma'$ where $\sigma' \in \text{SP.STATE}(\vec{v}, A)$

or $\mathfrak{M}_A(S)(\sigma) \uparrow$.

Thus, we may restrict the state transformation by writing

$$\mathfrak{M}_A(S)\colon \text{SP.STATE}(\vec{v}, A) \dashrightarrow \text{SP.STATE}(\vec{v}, A)$$

and build a representation of this mapping on A. Again let us emphasize that this fundamental simplification of the set of 'while' program computations to be analysed is achieved by Proposition 1 in 2.2.10, which eliminates our need to consider $\mathbb{u}$ and ε. (The Observation in 4.4.1 also eliminates the error state when considering functional programs in general, of course; at the moment we are examining the case of an *arbitrary* 'while' program S.)

For SP.STATE$(\vec{v}, A)$ we define the representation space $\boldsymbol{R}[\vec{v}, A]$ by

$$\begin{aligned} \boldsymbol{R}[\vec{v}, A] &= A_{k_1} \times \cdots \times A_{k_M} \\ &= A[\vec{k}] \end{aligned}$$

and code the states by means of the projection

$$\pi = \pi(\vec{v}, A)\colon \text{SP.STATE}(\vec{v}, A) \longrightarrow \boldsymbol{R}[\vec{v}, A]$$

defined by $$\pi(\sigma) = (\sigma(v_1), \ldots, \sigma(v_M)).$$

By the Corollary in 2.2.8, the state transformation $\mathcal{M}_A(S)$ may now be represented by the unique mapping $T_A(S)$ which commutes the following diagram:

$$\begin{array}{ccc} \text{SP.STATE}(\vec{v}, A) & \xrightarrow{\mathcal{M}_A(S)} & \text{SP.STATE}(\vec{v}, A) \\ \Big\downarrow \pi & & \Big\downarrow \pi \\ \boldsymbol{R}[\vec{v}, A] & \xrightarrow{T_A(S)} & \boldsymbol{R}[\vec{v}, A] \end{array}$$

i.e., $T_A(S)(\pi(\sigma)) \simeq \pi(\mathcal{M}_A(S)(\sigma))$ for all $\sigma \in \text{SP.STATE}(\vec{v}, A)$. In particular,

$$T_A(S)(\vec{x}) \simeq \begin{cases} \vec{y} & \text{if for all } \sigma \in \text{PR.STATE}(A) \text{ with } \sigma(\vec{v}) = \vec{x}, \\ & \quad \mathcal{M}_A(S)(\sigma) \downarrow \sigma' \text{ and } \sigma'(\vec{v}) = \vec{y}; \\ \uparrow & \text{otherwise.} \end{cases}$$

Consider now the coordinate functions

$$T_A^1(S), \ldots, T_A^M(S)$$

of $T_A(S)$. The map $T_A^i(S)\colon \boldsymbol{R}[\vec{v}, A] \longrightarrow A_{k_i}$ calculates the value of v_i upon the termination of S. These functions faithfully represent on A the state

transformation $\mathcal{M}_A(S)$ of an *arbitrary* 'while' program S applied to specified states.

Clearly, on proving that this representation of an arbitrary 'while' program is inductively definable on A, uniformly over $\mathbb{K}$, we can deduce that the functional semantics of an arbitrary functional 'while' program is inductively definable on A, uniformly over $\mathbb{K}$.

The inductive definability of the general representation is the subject of the Basic Lemma in the next section. We are obliged to consider the arbitrary 'while' programs of the Basic Lemma, rather than the functional programs of our desired Theorem, because subprograms of functional programs need not themselves be functional programs, at least with respect to their given input-output variables: this ruins the the possibility of a proof by induction on the complexity of functional programs.

EXAMPLE. Consider the functional program $[S,(x,y),(x,y)]$, with

$$\begin{aligned} S \equiv\ & t := x; \\ & x := y; \\ & y := t \end{aligned}$$

which swaps the values of x and y on any set. However for the subprogram

$$\begin{aligned} S_0 \equiv\ & x := y; \\ & y := t \end{aligned}$$

the i/o program $[S_0,(x,y),(x,y)]$ is not functional.

4.4.3 'while' program computability implies induction scheme definability

THEOREM. *Let $f=\{f_A \mid A\in\mathbb{K}\}$ be a family of partial mappings of type $(n,m;\vec{k},\vec{l})$ that is computable by a functional* 'while' *program $[S,\vec{v},\vec{w}]$ uniformly over $\mathbb{K}$. Then f is definable by induction schemes $\vec{\alpha}$ uniformly over $\mathbb{K}$. Moreover, one can effectively calculate $\vec{\alpha}$ from $[S,\vec{v},\vec{w}]$.*

In the light of 4.4.2 this theorem will follow from the

BASIC LEMMA. *Let $\vec{v}=(v_1,\ldots,v_M)$ be a list of simple program variables. For each* 'while' *program S with $\mathbf{Var}(S)\subseteq\vec{v}$ there is an M-tuple $\vec{\alpha}$ of induction schemes such that for all $A\in\mathbb{K}$ and $\vec{x}\in\boldsymbol{R}[\vec{v},A]$*

$$\{\vec{\alpha}\}^A(\vec{x}) \simeq \boldsymbol{T}_A(S)(\vec{x}).$$

Moreover, one can effectively calculate $\vec{\alpha}$ from S.

In particular, given any functional 'while' program $[S', \vec{v}', \vec{x}']$ in the Basic Lemma, we have only to set $S \equiv S'$ and $\vec{v}$ to be a list of variables containing $\boldsymbol{Var}(S')$, $\vec{v}'$ and $\vec{w}'$ to deduce the theorem.

The proof of the Basic Lemma requires a representation of program term evaluation on $\boldsymbol{R}[\vec{v}, A]$.

4.4.4 Program term evaluation

The evaluation of a program term at a state is a union of functions

$$\wp_A^i : \boldsymbol{ProgTerm}_i \to (\mathrm{PR.STATE}(A) \to A_i^{\mathsf{u}})$$

over all sorts i (see 1.2.5). This fundamental process of term evaluation can be restricted to our localization $\mathrm{SP.STATE}(\vec{v}, A)$ as follows.

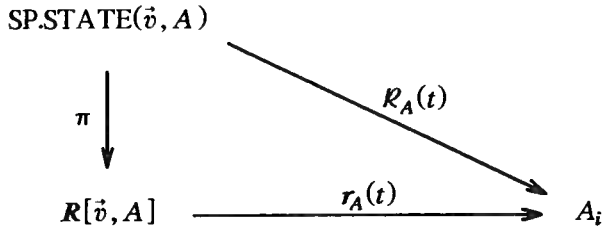
Let $\boldsymbol{ProgTerm}_i(\vec{v})$ denote the set of all program terms t of sort i with $\boldsymbol{ProgVar}(t) \subseteq \vec{v}$. Then, by Proposition 1 in 1.2.15, we can write

$$\wp_A^i : \boldsymbol{ProgTerm}_i(\vec{v}) \to (\mathrm{SP.STATE}(\vec{v}, A) \to A_i)$$

and make the obvious representation under $\pi = \pi(\vec{v}, A)$

$$r_A^i : \boldsymbol{ProgTerm}_i(\vec{v}) \to (\boldsymbol{R}[\vec{v}, A] \to A_i).$$

The function r_A, which is the union of the r_A^i over all sorts i, is uniquely determined by the condition that for all $t \in \boldsymbol{ProgTerm}_i(\vec{v})$ the following diagram commutes:



4.4.5 Inductive definability of program term evaluation

The following fact is easy to prove:

LEMMA. *For each sort i and for any program term $t^i \in \boldsymbol{ProgTerm}_i(\vec{v})$ there is an induction scheme α_{t^i} of type $(M; \vec{k}, i)$ such that for each $A \in \mathbb{K}$ and each $\vec{x} \in \boldsymbol{R}[\vec{v}, A]$,*

$$\{\alpha_{t^i}\}^A(\vec{x}) \simeq r_A(t^i)(\vec{x}).$$

PROOF. The proof is by induction on the complexity of program terms, simultaneously over the $r+2$ sorts $1, \ldots, r, \mathsf{N}, \mathsf{B}$ of Σ.

Basis. If t^i is the variable v_j of sort i, then we take as α_{t^i} the projection function scheme

$$<\mathrm{P}, n, \vec{k}, j>.$$

Induction step. There are two cases.

First suppose that $t^i \equiv F(t_1, \ldots, t_m)$, where F is of type $(m; i_1, \ldots, i_m, i)$. By the induction hypothesis there are induction schemes $\alpha_{t_1}, \ldots, \alpha_{t_m}$, with α_{t_j} of type $(m; \vec{k}, i_j)$, to define the subterms of t^i of various sorts. Thus we can take as α_{t^i} the scheme

$$<\mathrm{C}, n, m, \alpha_{t_1}, \ldots, \alpha_{t_m}, <\mathrm{O}, F, m, i_1, \ldots, i_m, i>>$$

being the composition of the subterm schemes with the basic operation scheme defined by F. It is easy to check that this α_{t^i} works.

The second case, namely that $t^i \equiv$ **if** b **then** t_1 **else** t_2 **fi**, is left as an exercise. □

Later, in 4.7.6, we will need a generalization of this fact to express, in terms of inductive definability, that the construction of α_t is uniform in t. This will make essential use of the scheme for simultaneous recursion, applied to all sorts.

4.4.6 Proof of Basic Lemma

We prove the Basic Lemma in 4.4.3 by induction on ***compl***(S).

Basis. $S \equiv v_i := t$. Observe that for $A \in \mathbb{K}$ and $\vec{x} \in \boldsymbol{R}[\vec{v}, A]$,

$$\boldsymbol{T}_A^i(S)(\vec{x}) \simeq \boldsymbol{r}_A(t)(\vec{x})$$

$$\boldsymbol{T}_A^j(S)(\vec{x}) \simeq x_j \quad \text{for } j = 1, \ldots, i-1, i+1, \ldots, M.$$

Now, using the Lemma in 4.4.5 and the projection schemes, the induction schemes $\vec{\alpha} = (\alpha_1, \ldots, \alpha_M)$ for $\{\boldsymbol{T}_A(S) \mid A \in \mathbb{K}\}$ are easy to construct.

Induction step. There are three cases:

Composition case, $S \equiv S_1; S_2$. By the induction hypothesis, there are M-tuples of schemes $\vec{\alpha}_1$ and $\vec{\alpha}_2$ to define the families of mappings

$$\{\boldsymbol{T}_A(S_1) \mid A \in \mathbb{K}\} \quad \text{and} \quad \{\boldsymbol{T}_A(S_2) \mid A \in \mathbb{K}\}$$

uniformly over $\mathbb{K}$. Clearly, for $A \in \mathbb{K}$ and $\vec{x} \in \boldsymbol{R}[\vec{v}, A]$,

$$T_A(S)(\vec{x}) \simeq (T_A(S_2) \circ T_A(S_1))(\vec{x})$$

and induction schemes $\vec{\alpha}$ can be made from $\vec{\alpha}_1$ and $\vec{\alpha}_2$, by composition, to define $\{T_A(S) \mid A \in \mathbb{K}\}$ uniformly over $\mathbb{K}$.

Conditional case, $S \equiv$ **if** b **then** S_1 **else** S_2 **fi**. By the induction hypothesis, there are M-tuples of schemes $\vec{\alpha}_1$ and $\vec{\alpha}_2$ to define

$$\{T_A(S_1) \mid A \in \mathbb{K}\} \quad \text{and} \quad \{T_A(S_2) \mid A \in \mathbb{K}\}$$

uniformly over $\mathbb{K}$. Clearly, for $A \in \mathbb{K}$ and $\vec{x} \in \boldsymbol{R}[\vec{v}, A]$,

$$T_A(S)(\vec{x}) \simeq \begin{cases} T_A(S_1)(\vec{x}) & \text{if } r_A(b)(\vec{x}) = \mathbb{t} \\ T_A(S_2)(\vec{x}) & \text{if } r_A(b)(\vec{x}) = \mathbb{f} \end{cases}$$

and induction schemes $\vec{\alpha}$ can be made from $\vec{\alpha}_1$ and $\vec{\alpha}_2$, by the Lemma in 4.4.5 and definition by cases, to define $\{T_A(S) \mid A \in \mathbb{K}\}$ uniformly over $\mathbb{K}$.

Iteration case, $S \equiv$ **while** b **do** S_0 **od**. By the induction hypothesis, there is an M-tuple of schemes $\vec{\alpha}_0$ to define $\{T_A(S_0) \mid A \in \mathbb{K}\}$ uniformly over $\mathbb{K}$. By the Iteration Lemma 4.2.4 there is an M-tuple of schemes $\vec{\alpha}_I$ which defines, uniformly over $\mathbb{K}$, the iteration mappings

$$I_A(S_0)(z, \vec{x}) \simeq (T_A(S_0) \circ \cdots \circ T_A(S_0))(\vec{x}) \qquad (z \text{ times})$$

for $\vec{x} \in \boldsymbol{R}[\vec{v}, A]$.

Consider now the "length of loop" function $L_A(S)\colon \boldsymbol{R}[\vec{v}, A] \to \mathbb{N}$ defined by

$$L_A(S)(\vec{x}) \simeq (\mu z \in \mathbb{N})[r_A(b)(I_A(S_0)(z, \vec{x})) \simeq \mathbb{f}].$$

The family $\{L_A(S) \mid A \in \mathbb{K}\}$ is definable by an induction scheme α_L, using Lemma 4.4.5, the tools of 4.2.2 and the least number scheme.

Finally, it remains to observe that for $A \in \mathbb{K}$ and $\vec{x} \in \boldsymbol{R}[\vec{v}, A]$,

$$T_A(S)(\vec{x}) \simeq I_A(S_0)(L_A(S)(\vec{x}), \vec{x})$$

and clearly the family $\{T_A(S) \mid A \in \mathbb{K}\}$ is uniformly definable over $\mathbb{K}$ by an M-tuple of induction schemes. □

4.5 COURSE-OF-VALUES INDUCTION

The equivalence of the computational power of 'while' programs (and recursive programs) and induction schemes over $\mathbb{K}$ suggests the question, "What is a function-theoretic equivalent of 'while'-array programs?" We answer that question with the concept of a *course-of-values induction scheme* which strengthens the power of primitive recursion on the standard numerical domain $\mathbb{N}$ on A to a *course-of-values recursion* on $\mathbb{N}$. Combinatorially, in primitive recursion on $\mathbb{N}$ the value $f(z,\vec{x})$ of a function $f: \mathbb{N}\times A[k] \to A_l$ depends on the single previous value $f(z\dot{-}1,\vec{x})$; in course-of-values recursion on $\mathbb{N}$ the value $f(z,\vec{x})$ depends on *several* earlier values

$$f(\delta_1(z,\vec{x}),\vec{x}),\ldots,f(\delta_d(z,\vec{x}),\vec{x})$$

where $\delta_i(z,\vec{x}) < z$ for $1 \le i \le d$. In the form of course-of-values induction we use, d is fixed for each function definition and is called the *degree* of the induction. (Unbounded *degrees* are possible, and other features to do with the algebraic parameters $\vec{x}$: see Section 4.8.) In the conventional situation of recursion on $\mathbb{N}$, course-of-values recursion of all degrees can be reduced to primitive recursion on $\mathbb{N}$ (see Kleene [1952], §46). In this section we consider this new mechanism, after the pattern of Section 4.1.

4.5.1 Functions computable by course-of-values induction on A

Consider the set CIND(A) of all partial functions on A obtained from the six devices of 4.1.1, but with the operation of simultaneous primitive recursion V replaced with the following:

V. *Simultaneous course-of-values recursion on* $\mathbb{N}$. Let $g_1,\ldots,g_m \in$ CIND(A) be functions with g_i of type $(n;\vec{k},l_i)$ for $i=1,\ldots,m$ and let $h_1,\ldots,h_m \in$ CIND(A) be functions with h_i of type

$$(1+n+dm;\mathbf{N},\vec{k},\vec{l},\ldots,\vec{l},l_i) \qquad (d \text{ times } \vec{l})$$

for $i=1,\ldots,m$. Let $\delta_1,\ldots,\delta_d \in$ CIND(A) be functions of type $(1+n;\mathbf{N},\vec{k},\mathbf{N})$, and now define $\hat{\delta}_1,\ldots,\hat{\delta}_d$ to be functions of the same type defined by

$$\hat{\delta}_i(z,\vec{x}) \simeq \min(\delta_i(z,\vec{x}),\, z\dot{-}1)$$

for $z\in\mathbb{N}, \vec{x}\in A[\vec{k}]$ and $i=1,\ldots,d$. Thus for any δ_i the mapping $\hat{\delta}_i$ is *contractive* in the sense that for all $z>0$, $\vec{x}\in A[\vec{k}]$

$$\hat{\delta}_i(z,\vec{x}) < z\,.$$

Then the functions $f_1, \ldots, f_m$ defined by

$$\begin{aligned} f_1(0,\vec{x}) &\simeq g_1(\vec{x}) \\ &\vdots \\ f_m(0,\vec{x}) &\simeq g_m(\vec{x}) \end{aligned}$$

and for $z > 0$,

$$\begin{aligned} f_1(z,\vec{x}) &\simeq h_1(z,\vec{x}, f_1(\hat{\delta}_1(z,\vec{x}),\vec{x}), \ldots, f_m(\hat{\delta}_1(z,\vec{x}),\vec{x}), \ldots \\ &\qquad \ldots, f_1(\hat{\delta}_d(z,\vec{x}),\vec{x}), \ldots, f_m(\hat{\delta}_d(z,\vec{x}),\vec{x})) \\ &\vdots \\ f_m(z,\vec{x}) &\simeq h_m(z,\vec{x}, f_1(\hat{\delta}_1(z,\vec{x}),\vec{x}), \ldots, f_m(\hat{\delta}_1(z,\vec{x}),\vec{x}), \ldots \\ &\qquad \ldots, f_1(\hat{\delta}_d(z,\vec{x}),\vec{x}), \ldots, f_m(\hat{\delta}_d(z,\vec{x}),\vec{x})) \end{aligned}$$

are functions in CIND(A) with f_i of type $(1+n; \mathsf{N}, \vec{k}, l_i)$.

The set CIND(A) is the set of all partial functions of the form $f : A[\vec{k}] \rightarrow A_l$ which are *course-of-values inductively definable* or *course-of-values inductively computable* on A; we abbreviate this terminology by *cov inductive definability* or *cov inductive computability*.

Clearly, IND(A) is a subset of CIND(A), since the primitive recursion scheme is a special case of the course-of-values recursion scheme (with degree 1 and $\delta(z,\vec{x}) = z \dot{-} 1$).

Following 4.1.3, we say that a partial (vector-valued) mapping $f : A[\vec{k}] \rightarrow A[\vec{l}]$ is *cov inductively definable* or *cov inductively computable* if its coordinate functions f_i are in CIND(A).

4.5.2 Cov induction schemes and their semantics

To properly define computability by cov induction we must define appropriate syntactic *cov induction schemes* and their semantics over the class $\mathbb{K}$. Since cov induction generalizes only the primitive recursion on $\mathbb{N}$ clause of induction, the necessary set $\boldsymbol{CIndSch} = \boldsymbol{CIndSch}(\Sigma)$ of cov induction schemes over signature Σ is defined by replacing clause V in the definition of induction schemes in 4.1.5 with the following clause:

V. For any cov induction schemes $\alpha_1, \ldots, \alpha_m$ with α_i of type $(n; \vec{k}, l_i)$, and cov induction schemes $\beta_1, \ldots, \beta_m$ with β_i of type $(1+n+dm; \mathsf{N}, \vec{k}, \vec{l}, \ldots, \vec{l}, l_i)$, and cov induction schemes $\Delta_1, \ldots, \Delta_d$ with Δ_i of type $(1+n; \mathsf{N}, \vec{k}, \mathsf{N})$, the notation

$$< \mathrm{CR}, n, m, d, i, \alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_m, \Delta_1, \ldots, \Delta_d >$$

is a cov induction scheme of type $(1+n; \mathsf{N}, \vec{k}, l_i)$.

The meaning of a cov induction scheme α of type $(m; \vec{k}, l)$ on the class $\mathbb{K}$ is a family of mappings

$$\mathcal{M}_A(\alpha): \ A[\vec{k}] \to A_l$$

indexed by $A \in \mathbb{K}$; that is the semantics of α is of the form

$$\mathcal{M}_{\mathbb{K}}(\alpha) = \{\mathcal{M}_A(\alpha) \mid A \in \mathbb{K}\}.$$

The semantics of schemes on A is informally described by the defining clauses for CIND(A); the formal definition of the meaning of a cov induction scheme α on the structure A is made by induction on the structure of α after the fashion of 4.1.6. Of course, a change is made at the cov recursion clause V, and to define $\mathcal{M}_A(\alpha)$ for

$$\alpha \equiv < \mathrm{CR}, n, m, d, i, \alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_m, \Delta_1, \ldots, \Delta_d >$$

we need the following fact to replace the Lemma in 4.1.6:

LEMMA. *Let $A \in \mathbb{K}$. Let $g_1, \ldots, g_m$ be partial functions on A with g_i of type $(n; \vec{k}, l_i)$ and let $h_1, \ldots, h_m$ be partial functions on A with h_i of type $(1+n+dm; \mathbf{N}, \vec{k}, \vec{l}, \ldots, \vec{l}, l_i)$. Let $\delta_1, \ldots, \delta_d$ be partial functions of type $(1+n; \mathbf{N}, \vec{k}, \mathbf{N})$, and define $\hat{\delta}_1, \ldots, \hat{\delta}_d$ by*

$$\hat{\delta}_i(z, \vec{x}) \simeq \min(\delta_i(z, \vec{x}), z \dot{-} 1)$$

for $z \in \mathbb{N}$, $\vec{x} \in A[\vec{k}]$ and $i = 1, \ldots, d$. Then there exists a unique *m-tuple of partial functions $\vec{f} = (f_1, \ldots, f_m)$ on A with f_i of type $(1+n; \mathbf{N}, \vec{k}, l_i)$ such that*

$$\begin{aligned} f_i(0, \vec{x}) &\simeq g_i(\vec{x}) \\ f_i(z, \vec{x}) &\simeq h_i(z, \vec{x}, f_1(\hat{\delta}_1(z, \vec{x}), \vec{x}), \ldots, f_m(\hat{\delta}_1(z, \vec{x}), \vec{x}), \ldots \\ &\qquad \ldots, f_1(\hat{\delta}_d(z, \vec{x}), \vec{x}), \ldots, f_m(\hat{\delta}_d(z, \vec{x}), \vec{x})) \end{aligned}$$

for all $z > 0$, $\vec{x} \in A[\vec{k}]$ and $i = 1, \ldots, m$.

4.5.3 Formal definition of computability over $\mathbb{K}$

A family $f = \{f_A \mid A \in \mathbb{K}\}$ of functions of type $(n; \vec{k}, l)$ is said to be *cov inductively definable uniformly over* $\mathbb{K}$ if there exists a cov induction scheme α of type $(n; \vec{k}, l)$ such that for every $A \in \mathbb{K}$ and every $\vec{x} \in A[\vec{k}]$,

$$f_A(\vec{x}) \simeq \{\alpha\}^A(\vec{x}).$$

The class of all such functions we denote CIND($\mathbb{K}$). And a family $f = \{f_A \mid A \in \mathbb{K}\}$ of (vector-valued) partial mappings of type $(n, m; \vec{k}, \vec{l})$ is said to be *cov inductively definable uniformly over* $\mathbb{K}$ if there exists a list

of cov induction schemes $\vec{\alpha} \equiv (\alpha_1, \ldots, \alpha_m)$, with α_i of type $(n; \vec{k}, l_i)$, uniformly defining the families $f_i = \{f_A^i \mid A \in \mathbb{K}\}$ of the coordinate functions of f.

4.5.4 Some important properties

The basic properties of inductive definability, proved in Section 4.2, are also true of cov inductive definability and we will have need of them in the following sections. Recall that these results concerned

the partial recursive and primitive recursive functions on $\mathbb{N}$;
the manipulation of arguments;
general definition-by-cases;
iteration of functions.

The statements of the theorems corresponding with the cov inductively definable functions follow *mutatis mutandis*. For reference, we write out the last two, but leave the proofs as exercises.

4.5.5 General definition by cases for cov inductive definability

PROPOSITION. *Let $\alpha_1, \ldots, \alpha_m$ be cov induction schemes of type $(n; \vec{k}, l)$ and let $\beta_1, \ldots, \beta_m$ be cov induction schemes of type $(n; \vec{k}, \mathsf{B})$. Then there is a cov induction scheme γ of type $(n; \vec{k}, l)$ such that for every $A \in \mathbb{K}$, $\vec{x} \in A[\vec{k}]$*

$$\{\gamma\}^A(\vec{x}) \simeq \begin{cases} \{\alpha_1\}^A(\vec{x}) & \text{if } \{\beta_1\}^A(\vec{x}) \downarrow \mathbb{1} \\ \{\alpha_2\}^A(\vec{x}) & \text{if } \{\beta_1\}^A(\vec{x}) \downarrow \mathbb{1} \text{ and } \{\beta_2\}^A(\vec{x}) \downarrow \mathbb{1} \\ \quad \vdots & \\ \{\alpha_m\}^A(\vec{x}) & \text{if } \{\beta_i\}^A(\vec{x}) \downarrow \mathbb{1} \text{ for } 1 \leq i < m \\ & \quad \text{and } \{\beta_m\}^A(\vec{x}) \downarrow \mathbb{1}. \end{cases}$$

Notice that $\{\gamma\}^A(\vec{x}) \uparrow$ if, and only if, for some $1 \leq i \leq m$,

$$\{\alpha_j\}^A(\vec{x}) \downarrow \mathbb{1} \text{ for } j \leq i \text{ and } \{\alpha_i\}^A(\vec{x}) \uparrow.$$

We will have particular need of the following kinds of results, not included in 4.2.3:

COROLLARY 1 (Definition by primitive recursive cases). *Let $\alpha_1, \ldots, \alpha_m$ be cov induction schemes of type $(n; \vec{k}, l)$. Let $R_1, \ldots, R_m$ be primitive recursive subsets of $\mathbb{N}$ which are pairwise disjoint and mutually exhaustive, i.e.*

$$R_i \cap R_j = \emptyset \quad \text{and} \quad \bigcup_{i=1}^{m} R_i = \mathbb{N}.$$

Then there is a cov induction scheme γ *of type* $(1+n; \mathsf{N}, \vec{k}, l)$ *such that for every* $A \in \mathbb{K}$, $z \in \mathbb{N}$, $\vec{x} \in A[\vec{k}]$,

$$\{\gamma\}^A(z, \vec{x}) \simeq \begin{cases} \{\alpha_1\}^A(\vec{x}) & \text{if } z \in R_1 \\ \vdots & \\ \{\alpha_m\}^A(\vec{x}) & \text{if } z \in R_m . \end{cases}$$

PROOF. By the Proposition and Theorem 4.2.1. □

Now the next corollary extends the form of the course-of-values recursion scheme by definition by cases; it has important work to do in Section 4.7 (and was needed in 3.6.4).

COROLLARY 2 (Course-of-values recursion with definition by cases). *Let* $g_1, \ldots, g_m \in \mathrm{CIND}(A)$ *be partial functions with* g_i *of type* $(n; \vec{k}, l_i)$ *for* $i = 1, \ldots, m$, *and let*

$$h_1^1, \ldots, h_1^p \ , \ldots, \ h_m^1, \ldots, h_m^p \in \mathrm{CIND}(A)$$

where for each $i = 1, \ldots, m$ *the functions* $h_i^1, \ldots, h_i^p$ *are of type*

$$(1+n+dm; \mathsf{N}, \vec{k}, \vec{l}, \ldots, \vec{l}, l_i).$$

Let $\delta_1, \ldots, \delta_d \in \mathrm{CIND}(A)$. *Let* $R_1, \ldots, R_p$ *be primitive recursive subsets of* $\mathbb{N}$ *which are pairwise disjoint and mutually exhaustive, i.e.*

$$R_i \cap R_j = \emptyset \quad \text{and} \quad \bigcup_{i=1}^{p} R_i = \mathbb{N}.$$

Then the functions $f_1, \ldots, f_m$ *defined by*

$$f_i(0, \vec{x}) \simeq g_i(\vec{x})$$

and for $z > 0$

$$f_i(z, \vec{x}) \simeq \begin{cases} h_i^1(z, \vec{x}, \hat{\delta}_1(z, \vec{x}), \ldots, \hat{\delta}_d(z, \vec{x})) & \text{if } z \in R_1 \\ \vdots & \\ h_i^p(z, \vec{x}, \hat{\delta}_1(z, \vec{x}), \ldots, \hat{\delta}_d(z, \vec{x})) & \text{if } z \in R_p \end{cases}$$

are functions in $\mathrm{CIND}(A)$ *with* f_i *of type* $(1+n; \mathsf{N}, \vec{k}, l_i)$.

PROOF. Combine $h_i^1, \ldots, h_i^p$ into a single h_i by means of Corollary 1. □

4.5.6 Iteration lemma for cov inductive definability

ITERATION LEMMA. *Let* $f=\{f_A \mid A\in\mathbb{K}\}$ *be a family of partial mappings of type* $(n,n;\vec{k},\vec{k})$ *that is uniformly definable over* $\mathbb{K}$ *by cov induction schemes* $\vec{\alpha}$. *Then the family* $I=\{I_A \mid A\in\mathbb{K}\}$ *of iteration maps*

$$I_A:\ \mathbb{N}\times A[\vec{k}]\to A[\vec{k}]$$

defined by

$$I_A(z,\vec{x}) \simeq (f_A\circ\cdots\circ f_A)(\vec{x}) \qquad (z \text{ times})$$
$$\simeq f_A^z(\vec{x})$$

is uniformly definable over $\mathbb{K}$ *by cov induction schemes* $\vec{\beta}$.

4.5.7 Reduction of cov recursion to primitive recursion

Under certain circumstances, definition of partial functions by cov recursion can be replaced by definition by primitive recursion. This is, in fact, the case when (1) we are working in A^*, *and* (2) the ranges of the recursively defined functions (denoted $l_1,\ldots,l_m$ in 4.5.1) are all of *unstarred* types. When these two conditions are satisfied, the cov recursion can be coded by primitive recursion, with range types $l_1^*,\ldots,l_m^*$. The idea is that at any argument z, the sequence of values $f_i(0,\vec{x}),\ldots,f_i(z-1,\vec{x})$, all in $A_{l_i}^{\mathrm{u}}$, can be coded by a single element of $A_{l_i}^*$ (for $i=1,\ldots,m$).

The technique is just as in Kleene [1952], §46, for the case of recursion over the natural numbers, and details are omitted.

4.6 FROM COV INDUCTION SCHEMES TO 'WHILE'-ARRAY PROGRAMS

We will prove that a family of functions definable by cov induction schemes over $\mathbb{K}$ is computable by a functional **while**-array program over $\mathbb{K}$ — a basic companion theorem for Theorem 4.3.4. The preparatory discussions of how functions may be computed by programs, in 4.3.1 and 4.3.2, apply to **'while'**-array programs, and hence we need only register the following definition before proving the theorem.

4.6.1 'while'-array computable functions

A family $f = \{f_A \mid A \in \mathbb{K}\}$ of partial functions of type $(n, m; \vec{k}, \vec{l})$ is said to be *computable by a* 'while'*-array program uniformly over* $\mathbb{K}$ if there is an appropriate functional 'while'-array program $[S, \vec{v}, \vec{w}]$ such that for all $A \in \mathbb{K}$ and all $\vec{x} \in A[\vec{k}]$,

$$f_A(\vec{x}) \simeq \mathit{fn}_A(S, \vec{v}, \vec{w})(\vec{x}).$$

4.6.2 Cov induction scheme definability implies 'while'-array program computability

THEOREM. *Let* $f = \{f_A \mid A \in \mathbb{K}\}$ *be a family of partial mappings of type* $(n, m; \vec{k}, \vec{l})$ *that is definable by cov induction schemes* $\vec{\alpha}$ *uniformly over* $\mathbb{K}$. *Then* f *is computable by a functional* 'while'*-array program* $[S, \vec{v}, \vec{w}]$ *uniformly over* $\mathbb{K}$. *Moreover, one can effectively calculate* $[S, \vec{v}, \vec{w}]$ *from* $\vec{\alpha}$.

PROOF. Recall to mind the proof of Theorem 4.3.4. Again we first consider coordinate functions and construct a functional 'while'-array program $[S, \vec{v}, w]$ for each cov induction scheme α of type $(n; \vec{k}, l)$. The construction is by induction on the definition of α.

The basis cases are exactly as in 4.3.4; and the induction step cases of composition schemes and least number search schemes are as in 4.3.4, too. We have only to consider the case of cov induction,

$$\alpha \equiv < \mathrm{CR}, n, m, d, i, \alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_m, \Delta_1, \ldots, \Delta_d >$$

where α_i is of type $(n; \vec{k}, l_i)$, β_i is of type $(1+n+dm; \mathsf{N}, \vec{k}, \vec{l}, \ldots, \vec{l}, l_i)$, and Δ_i is of type $(1+n; \mathsf{N}, \vec{k}, \mathsf{N})$.

By the induction hypothesis, there exist functional 'while'-array programs

$$[S_i, \vec{v}_i, w_i]$$
$$[\hat{S}_i, (\vec{y}_i, b_i, \vec{z}_{i1}, \ldots, \vec{z}_{id}), s_i]$$
$$[\tilde{S}_j, (\vec{p}_j, q_j), e_j]$$

which compute α_i, β_i and Δ_j over $\mathbb{K}$. We assume *these programs have no variables in common.* Consider the program S depicted in Figure 1, with new simple variables $c, b, \vec{v}, \vec{r}, \hat{e}_1, \ldots, \hat{e}_d$ and new array variables $a_1, \ldots, a_m$. In S it is the task of $a_i[\hat{e}_j]$ to store the value $f_i(\hat{\delta}_j(c, \vec{x}), \vec{x})$ as $f(c, \vec{x})$ is computed "bottom-up".

We leave as an exercise for the reader the task of verifying that $[S, (\vec{v}, b), r_i]$ is a functional program that computes the cov induction scheme α over $\mathbb{K}$.

$S \equiv \vec{v}_1 := \vec{v}\,;\ S_1\,;\ r_1 := w_1;$

$\vdots$

$\vec{v}_m := \vec{v}\,;\ S_m\,;\ r_m := w_m\,;$

$c := 0;$

while $c < b$

do $a_1[c] := r_1\,; \ldots ; a_m[c] := r_m\,;$

$c := c+1;$

$$\left\{\begin{array}{l} \vec{p}_1 := \vec{v};\ q_1 := c;\ \tilde{S}_1\,;\ \hat{e}_1 := \text{if } e_1 \geq c \text{ then } c-1 \text{ else } e_1 \text{ fi}; \\ \qquad\qquad \vdots \\ \vec{p}_d := \vec{v};\ q_d := c;\ \tilde{S}_d\,;\ \hat{e}_d := \text{if } e_d \geq c \text{ then } c-1 \text{ else } e_d \text{ fi}; \end{array}\right\}$$

$$\left\{\begin{array}{l} \vec{y}_1 := \vec{v};\ b_1 := c; \\ \vec{z}_{11} := (a_1[\hat{e}_1], \ldots, a_m[\hat{e}_1]); \\ \qquad\qquad \vdots \\ \vec{z}_{1d} := (a_1[\hat{e}_d], \ldots, a_m[\hat{e}_d]); \\ \hat{S}_1\,; \end{array}\right\}$$

$\vdots$

$$\left\{\begin{array}{l} \vec{y}_m := \vec{v};\ b_m := c; \\ \vec{z}_{m1} := (a_1[\hat{e}_1], \ldots, a_m[\hat{e}_1]); \\ \qquad\qquad \vdots \\ \vec{z}_{md} := (a_1[\hat{e}_d], \ldots, a_m[\hat{e}_d]); \\ \hat{S}_m\,; \end{array}\right\}$$

$r_1 := s_1\,; \ldots ; r_m := s_m$

od

FIGURE 1

The argument to show that this method can be applied to the cov induction schemes $\alpha_1, \ldots, \alpha_m$ for the coordinate mappings of f, and that the resulting functional programs $[S_1, \vec{v}_1, w_1], \ldots, [S_m, \vec{v}_m, w_m]$ can be combined to make a single functional program $[S, \vec{v}, \vec{w}]$ to compute f, follows precisely the argument in 4.3.4, and we take the liberty of omitting it. □

4.7 FROM 'WHILE'-ARRAY PROGRAMS TO COV INDUCTION SCHEMES

We will prove the converse of Theorem 4.6.2 by simulating functional 'while'-array program computation on STATE(A) with cov inductively definable functions on A, after the fashion of the simulation seen in Section 4.4. The presence of arrays complicates the construction of the representation space and transformation, and we must again consider the elimination of ε and $\sqcup$ with care. More importantly, the argument that the new representation maps are cov inductively definable requires ingenuity and a good deal more coding machinery.

4.7.1 A standard representation of the state space and state transformations

Recalling the plan of 4.4.2, we will first represent the state transformation

$$\mathcal{M}_A(S)\colon \text{PR.STATE}(A) \dashrightarrow \text{STATE}(A)$$

of a 'while'-array program S as a partial mapping $T_A^*(S)$ on A^*. For this $T_A^*(S)$ we will later make a representation on A, by means of a partial mapping $G_A(S)$.

Let S be an arbitrary 'while'-array program whose variables are among the simple variables $\vec{v} = (v_1, \ldots, v_{M_1})$ of sorts $\vec{k} = (k_1, \ldots, k_{M_1})$ and the array variables $\vec{a} = (a_1, \ldots, a_{M_2})$ of sorts $\vec{l} = (l_1, \ldots, l_{M_2})$ respectively; in symbols, $\mathit{Var}(S) \subseteq \vec{v} \cup \vec{a}$. On considering the effect of executing S on the specified states

$$\text{SP.STATE}(\vec{v}, A) =_{df} \{\sigma \in \text{PR.STATE}(A) \mid \text{for } 1 \leq i \leq M_1,\ \sigma(v_i) \neq \sqcup\}$$

we observe that, by Proposition 2 in 2.2.10 and in partial analogy to 4.4.2, the state transformation may be restricted thus:

$$\mathcal{M}_A(S)\colon \text{SP.STATE}(\vec{v}, A) \dashrightarrow \text{SP.STATE}(\vec{v}, A) \cup \{\varepsilon\}.$$

The reason for the possibility of S converging to an error state is that S may invoke an unspecified intermediate variable $<a_i, j>$: contrast Propositions 1 and 2 in 2.2.10. If S is a functional program with respect to certain of its simple variables, then this error termination will not occur, of course (recall 4.4.1 and see 4.7.2). However, we must follow the pattern of 4.4.2 and continue to examine a general 'while'-array program.

We next represent SP.STATE$(\vec{v}, A)$ over A^* by means of the new representation space

$$\begin{aligned} \boldsymbol{R}[\vec{v}, \vec{a}, A] &= A_{k_1} \times \cdots \times A_{k_{M_1}} \times A^*_{l_1} \times \cdots \times A^*_{l_{M_2}} \\ &= A[\vec{k}, \vec{l}^*] \end{aligned}$$

using the projection

$$\pi^* = \pi^*(\vec{v}, \vec{a}, A) : \text{SP.STATE}(\vec{v}, A) \longrightarrow \boldsymbol{R}[\vec{v}, \vec{a}, A]$$

defined by $\pi^*(\sigma) = (x_1, \ldots, x_{M_1}, \xi_1, \ldots, \xi_{M_2}) = (\vec{x}, \vec{\xi})$

where for $i = 1, \ldots, M_1$, $x_i = \sigma(v_i)$ and for $i = 1, \ldots, M_2$, $\xi_i(j) = \sigma(a_i, j)$ (compare 2.6.5).

In addition, we adjoin to $\boldsymbol{R}[\vec{v}, \vec{a}, A]$ a special error vector

$$\vec{\varepsilon} =_{df} (\varepsilon_{k_1}, \ldots, \varepsilon_{k_{M_1}}, \varepsilon^*_{l_1}, \ldots, \varepsilon^*_{l_{M_2}})$$

where ε_{k_i} and $\varepsilon^*_{l_j}$ are new objects associated with A_{k_i} and $A^*_{l_j}$ respectively. Let π^*_ε be π^* extended by mapping the error state ε to the error vector $\vec{\varepsilon}$.

By the Corollary in 2.2.8, the state transformation $\mathcal{M}_A(S)$ may now be represented by the unique mapping $T^*_A(S)$ which commutes the following diagram (compare 4.4.2):

$$\begin{array}{ccc} \text{SP.STATE}(\vec{v}, A) & \xrightarrow{\mathcal{M}_A(S)} & \text{SP.STATE}(\vec{v}, A) \cup \{\varepsilon\} \\ \Big\downarrow \pi^* & & \Big\downarrow \pi^*_\varepsilon \\ \boldsymbol{R}[\vec{v}, \vec{a}, A] & \xrightarrow{T^*_A(S)} & \boldsymbol{R}[\vec{v}, \vec{a}, A] \cup \{\vec{\varepsilon}\} \end{array}$$

i.e. $T^*_A(S)(\pi^*(\sigma)) \simeq \pi^*_\varepsilon(\mathcal{M}_A(S)(\sigma))$ for all $\sigma \in \text{SP.STATE}(\vec{v}, A)$.

In particular, consider the coordinate functions

$$T_A^*(S)_1, \ldots, T_A^*(S)_{M_1}, T_A^*(S)_{M_1+1}, \ldots, T_A^*(S)_{M_1+M_2}$$

of $T_A^*(S)$. We rewrite the first M_1 mappings as the family

$$T_{v_i}^*(S)\colon\ R[\vec{v},\vec{a},A] \to A_{k_i} \cup \{\varepsilon_{k_i}\}$$

which calculates the value of v_i upon the termination of S. And we change the next M_2 mappings to the family

$$T_{a_i}^*(S)\colon\ R[\vec{v},\vec{a},A] \times \mathbb{N} \to A_{l_i}^{\mu} \cup \{\varepsilon_{l_i}^*\}$$

where

$$T_{a_i}^*(S)(\vec{x},\vec{\xi},j) \simeq (T_A^*(S)_{M_1+i}(\vec{x},\vec{\xi}))(j).$$

The map $T_{a_i}^*(S)$ calculates the value of $<a_i, j>$ upon the termination of S. (We trust that the dropping of the reference to A will not lead to confusion.) These coordinate functions faithfully represent on A^*, with the error vector $\vec{\varepsilon}$ adjoined, the state transformation $\mathcal{M}_A(S)$ of an *arbitrary* 'while'-array program S.

4.7.2 A refinement for functional programs

If S is a functional program with respect to certain of its simple variables, we can further restrict the class of S computations to be considered to those computations with *initial states all of whose array variables are unspecified*, since the input-output behaviour is determined by the simple variables: recall the discussion in 4.4.1. Let us write

$$\text{SP.STATE}(\vec{v},\vec{a},A) = \{\sigma \in \text{SP.STATE}(\vec{v},A) \mid\ \sigma(a_i, j) = \text{ш} \text{ for } 1 \le i \le M_2 \text{ and } j \in \mathbb{N}\}.$$

Furthermore, if S is functional then the error state will not arise in a computation on this set of states, and hence we may restrict the state transformation thus:

$$\mathcal{M}_A(S)\colon\ \text{SP.STATE}(\vec{v},\vec{a},A) \dashrightarrow \text{SP.STATE}(\vec{v},A).$$

Let us consider the effect of these properties in the representation machinery just introduced.

The set of initial states can be represented by the restriction of π^* to

$$\pi\colon\ \mathrm{SP.STATE}(\vec{v},\vec{a},A) \longrightarrow \boldsymbol{R}[\vec{v},A]$$

defined by $$\pi(\sigma) = (\sigma(v_1),\ldots,\sigma(v_{M_1})) = \vec{x}$$

and the state transformation by the restriction of $T_A^*(S)$ to

$$T_A(S)\colon\ \boldsymbol{R}[\vec{v},A] \dashrightarrow \boldsymbol{R}[\vec{v},\vec{a},A]$$

defined by

$$T_A(S)(\vec{x}) \simeq T_A^*(S)(\vec{x}, \mathsf{Null}_{l_1}^A,\ldots,\mathsf{Null}_{l_{M_2}}^A)$$

wherein $\mathsf{Null}_{l_i}^A(j) = \mathbb{u}_{l_i} \in A_{l_i}^{\mathsf{u}}$ for $i = 1,\ldots,M_2$. Thus we have the following commutative diagram:

$$\begin{array}{ccc}
\mathrm{SP.STATE}(\vec{v},\vec{a},A) & \xrightarrow{\mathcal{M}_A(S)} & \mathrm{SP.STATE}(\vec{v},A) \\
\Big\downarrow \pi & & \Big\downarrow \pi^* \\
\boldsymbol{R}[\vec{v},A] & \xrightarrow{T_A(S)} & \boldsymbol{R}[\vec{v},\vec{a},A]
\end{array}$$

With these observations, eliminating A^* from the domain of $T_A^*(S)$ and $\vec{\varepsilon}$ from its range, and on applying the method of 4.7.1 to the coordinate functions of $T_A(S)$ to eliminate all mention of finite functions in the range, we obtain a representation of the behaviour of S on A and A^{u} by means of the mappings

$$T_{v_i}(S)\colon\ \boldsymbol{R}[\vec{v},A] \dashrightarrow A_{k_i}$$

$$T_{a_i}(S)\colon\ \boldsymbol{R}[\vec{v},A]\times\mathbb{N} \dashrightarrow A_{l_i}^{\mathsf{u}}$$

Thus, to prove that the functional semantics of a functional program is cov inductively definable, it is sufficient to prove that these functions are uniformly cov inductively definable on A and A^{u}. (Of course, it is sufficient to prove that the simple variable functions $T_{v_i}(S)$ are uniformly cov inductively definable on A).

However the proof presented in this section is not simply contained within the pleasant and now familiar world of definability on A, or even on A^{u}. Unfortunately, the technical machinery required must be made to deal with errors in the simulation of $T_A^*(S)$ for an *arbitrary* 'while'-array

program S; thus, the special error vector $\vec{\varepsilon}$ will haunt the proof as we must construct cov inductive functions on the structures A and A^{μ} with errors represented.

A reason for this state of affairs can be indicated here. The various representation maps for a functional program S are constructed by induction on the complexity of S. The subprograms of S need not be functional, however (recall 4.4.2). Thus, although S may be assumed to possess the desirable properties just discussed, its component 'while'-array programs may, on independent execution, give rise to error states. This problem did not occur in the case of 'while' programs because both the undefined element and errors could be eliminated from arbitrary 'while' program computations with our special set of initial states: see 4.4.2.

EXAMPLE. Consider the functional program $[S,c,d]$ with

$$\begin{aligned} S \equiv\ & a[0]:=0\,; \quad a[1]:=1\,; \\ & i:=2\,; \\ & \textbf{while } i\leq c \textbf{ do } a[i]:=a[i-2]+a[i-1]\,; \\ & \qquad\qquad\qquad i:=i+1 \\ & \qquad\qquad \textbf{od}\,; \\ & d:=a[c] \end{aligned}$$

which computes the Fibonacci sequence on $\mathbb{N}$. However, for the subprogram

$$S_0 \equiv d:=a[c]$$

the i/o program $[S_0,c,d]$ is not functional, and its execution will in general yield the error state.

4.7.3 'while'-array program computability implies cov induction scheme definability

THEOREM. *Let $f=\{f_A \mid A\in\mathbb{K}\}$ be a family of partial mappings of type $(n,m;\vec{k},\vec{l})$ that is computable by a functional 'while'-array program $[S,\vec{v},\vec{w}]$ uniformly over $\mathbb{K}$. Then f is definable by cov induction schemes $\vec{\alpha}$ uniformly over $\mathbb{K}$. Moreover, one can effectively calculate $\vec{\alpha}$ from $[S,\vec{v},\vec{w}]$.*

In view of 4.7.2, the theorem will follow from:

BASIC LEMMA. *Let* $\vec{v} = (v_1, \ldots, v_{M_1})$ *and* $\vec{a} = (a_1, \ldots, a_{M_2})$ *be lists of simple and array variables, respectively. For each functional* 'while'*-array program* S *with* $\mathbf{Var}(S) \subseteq \vec{v} \cup \vec{a}$ *there is an* M_1+M_2 *tuple of cov induction schemes* $\vec{\alpha}$ *over* Σ *and* $\vec{\beta}$ *over* $\Sigma^{\mathbb{u}}$ *such that for all* $A \in \mathbb{K}$ *and* $\vec{x} \in A[\vec{k}]$,

$$\{\alpha_i\}^A(\vec{x}) \simeq \boldsymbol{T}_{v_i}(S)(\vec{x})$$

$$\{\beta_i\}^{A^{\mathbb{u}}}(\vec{x}, j) \simeq \boldsymbol{T}_{a_i}(S)(\vec{x}, j).$$

Moreover, one can effectively calculate $\vec{\alpha}, \vec{\beta}$ *from* S.

We will distribute the proof over the remaining eight subsections.

4.7.4 Strategy of the proof

The representation $\boldsymbol{T}_A^*(S)$ on A^* of $\mathcal{M}_A(S)$ defined, for an arbitrary program S, in 4.7.1, will be partially represented by a mapping $\boldsymbol{G}_A(S)$ on A, that successfully employs coding on the domain $\mathbb{N}$ of A to simulate the effect of unbounded (algebraic) array memory in a 'while'-array program computation on A; and to simulate the effects of $\mathbb{u}$ and ε in the construction of $\boldsymbol{T}_A^*(S)$. The key to this new representation is the following elementary, yet important, fact, adapted from Tucker [1980].

LOCALIZATION LEMMA. *Let* S *be a* 'while'*-array program and let* $\sigma \in \mathrm{PR.STATE}(A)$. *Then the specified values of the variables of* S *at each state, occurring in the computation of* $\mathcal{M}_A(S)(\sigma)$, *lie in the substructure* $\boldsymbol{sub}_{A,S}(\sigma)$ *of* A *generated by the values of the variables of* S *at the initial state* σ. *In particular, if* $\mathcal{M}_A(S)(\sigma)\downarrow\sigma'$ *then the values of the variables of* S *at the final state* σ' *lie in* $\boldsymbol{sub}_{A,S}(\sigma)$.

PROOF. By induction on $\boldsymbol{compl}(S)$. □

For any $\sigma \in \mathrm{SP.STATE}(\vec{v}, A)$, the computation of $\mathcal{M}_A(S)(\sigma)$ is represented by the computation of $\boldsymbol{T}_A^*(S)(\pi^*(\sigma))$ using

$$\boldsymbol{R}[\vec{v}, \vec{a}, A] = A_{k_1} \times \cdots \times A_{k_{M_1}} \times A_{l_1}^* \times \cdots \times A_{l_{M_2}}^*.$$

By the Localization Lemma, the computation for a given σ can be localized as a computation taking place over $\boldsymbol{sub}_{A,S}(\sigma)$ which is the substructure of A generated by the *finitely many* specified values $\vec{x}$ of $\vec{v}$ evaluated in A, and $\vec{\xi}$ of $\vec{a}$ evaluated in $A^{\mathbb{u}}$. In particular, $\boldsymbol{sub}_{A,S}(\sigma)$ is, in a more conventional algebraic notation,

$$\langle \vec{x}, \vec{\xi} \rangle = \langle \{x_1, \ldots, x_{M_1}\} \cup \bigcup_{i=1}^{M_2} \{\xi_i(j) \mid j \in \mathbb{N} \text{ and } \xi_i(j) \neq \mathbb{u}\} \rangle.$$

This means that, at a given state σ, the computation of $T_A^*(S)(\pi^*(\sigma))$ with $\pi^*(\sigma) = (\vec{x}, \vec{\xi})$ uses the localized structure

$$\boldsymbol{R}[\vec{v},\vec{a},A](\vec{x},\vec{\xi}) = <\vec{x},\vec{\xi}>_{k_1} \times \cdots \times <\vec{x},\vec{\xi}>_{k_{M_1}} \times <\vec{x},\vec{\xi}>_{l_1}^* \times \cdots \times <\vec{x},\vec{\xi}>_{l_{M_2}}^*,$$

wherein $<\vec{x},\vec{\xi}>_s$ is the domain of sort s in the many-sorted structure $<\vec{x},\vec{\xi}>$.

With the use of initial states $\sigma \in \text{SP.STATE}(\vec{v},\vec{a},A)$ in mind, we make a simplification by working with the representation of $T_A^*(S)$ on $\pi^*(\sigma)=(\vec{x},\vec{\xi})$ with $\vec{\xi}_i = \textbf{Null}_i^A$ (recall 4.7.2). Thus, the local subspace of interest is

$$\boldsymbol{R}[\vec{v},\vec{a},A](\vec{x}) = <\vec{x}>_{k_1} \times \cdots \times <\vec{x}>_{k_{M_1}} \times <\vec{x}>_{l_1}^* \times \cdots \times <\vec{x}>_{l_{M_2}}^*.$$

On constructing this space from the input $\pi^*(\sigma)=\vec{x}$, the computation can be unfolded on $\boldsymbol{R}[\vec{v},\vec{a},A](\vec{x})$.

Now the fact that $<\vec{x}>$ is a finitely generated structure prompts us to build an enumeration or coding of $<\vec{x}>$ in $\mathbb{N}$ that is uniform over all $\vec{x} \in A[\vec{k}]$. This is done using a Gödel numbering of terms and term evaluation and results in a map

$$\boldsymbol{te}_A : \mathbb{N} \times A[\vec{k}] \longrightarrow A$$

such that for all $\vec{x} \in A[\vec{k}]$

$$\boldsymbol{te}_A(\_,\vec{x}) : \mathbb{N} \longrightarrow <\vec{x}>$$

is a surjection.

This Gödelization of $<\vec{x}>$ leads to a coding of $\boldsymbol{R}[\vec{v},\vec{a},A](\vec{x})$ in $\mathbb{N}$ which is uniform over all $\vec{x} \in A[\vec{k}]$: we build a map

$$\Gamma : A[\vec{k}] \times \mathbb{N}^{M_1} \times \mathbb{N}^{M_2} \longrightarrow \boldsymbol{R}[\vec{v},\vec{a},A]$$

such that

$$\Gamma(\vec{x},\_,\_) : \mathbb{N}^{M_1} \times \mathbb{N}^{M_2} \longrightarrow \boldsymbol{R}[\vec{v},\vec{a},A](\vec{x})$$

is a surjection.

Thus, the strategy of the argument is to construct and use this machinery to show that the computation of $T_A^*(S)(\pi^*(\sigma))$ with $\pi^*(\sigma)=\vec{x}$ can be simulated on $\mathbb{N}$ uniformly in $\vec{x}$, by cov inductively definable mappings on A. More precisely, the reconstruction of the global behaviour of S, as represented by $T^*(S)$, from the localizations based on Γ is made by constructing a cov inductively definable map $G_A(S)$ on A that commutes the following diagram:

$$\begin{array}{ccc} \boldsymbol{R}[\vec{v},\vec{a},A] & \xrightarrow{T_A^*(S)} & \boldsymbol{R}[\vec{v},\vec{a},A] \cup \{\vec{\varepsilon}\} \\ \Big\uparrow \Gamma & & \Big\uparrow \Gamma_e \\ A[\vec{k}]\times\mathbb{N}^{M_1}\times\mathbb{N}^{M_2} & \xrightarrow{G_A(S)} & A[\vec{k}]\times\mathbb{N}^{M_1}\times\mathbb{N}^{M_2} \cup A[\vec{k}]\times\{e\} \end{array}$$

wherein e is an M_1+M_2 tuple coded within the domain $\mathbb{N}$ of A and used to represent the error vector, *i.e.* $\Gamma_e(\vec{x},e)=\vec{\varepsilon}$. Notice that $G_A(S)$ is not uniquely determined by the diagram. (Why?) The proof of the Basic Lemma in 4.7.3 is then easy to conclude.

In the next few sections we will construct Γ and G.

4.7.5 Enumerating finitely generated substructures

Each element y of the substructure $<\vec{x}>$ of A is obtained by finitely many applications of the operations of A on the generators $\vec{x}$. That is, each element y is the value of a term $t \in \boldsymbol{ProgTerm}(\vec{v})$ at $\vec{x}$ and, recalling the semantics of terms in 4.4.4, we can write $y = r_A(t)(\vec{x})$. The enumeration of the substructure $<\vec{x}>$ is made from a numerical coding of $\boldsymbol{ProgTerm}(\vec{v})$ and simple term evaluation.

Consider again the codings of syntax first discussed in 3.6.3. From those postulates we can derive a primitive recursive coding of $\boldsymbol{ProgTerm}(\vec{v})$ with the monotonicity property that the code $\ulcorner t \urcorner$ increases strictly with $\boldsymbol{compl}(t)$. We also assume that the customary operations on syntax are primitive recursive with respect to the coding; in particular the substitution of terms $t_1,\ldots,t_m$ of types $i_1,\ldots,i_m$ for the variables of a term t_0 of type l_0 to make a term

$$t_0(t_1/v_1,\ldots,t_n/v_n)$$

of type l_0 is primitive recursive. Suppose that γ denotes a right inverse of the coding, *i.e.* it is the union over all sorts i of maps

$$\gamma^i : \mathbb{N} \to \boldsymbol{ProgTerm}_i(\vec{v})$$

where, for $t \in \boldsymbol{ProgTerm}_i(\vec{v})$,

$$\gamma^i(\ulcorner t \urcorner) = t\,.$$

Next, we redefine the semantics of terms of 4.4.4 as the uniform *simple term evaluation* TE_A, which is the union over all sorts i of maps

$$TE_A^i\colon\ ProgTerm_i(\vec{v}) \times R[\vec{v}, A] \longrightarrow A_i\,,$$

by setting $$TE_A(t, \vec{x}) = r_A(t)(\vec{x})\,.$$

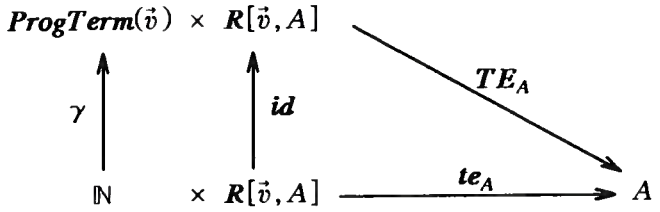
The construction of $<\vec{x}>$ can now be described by saying that $<\vec{x}>$ is the image of $ProgTerm(\vec{v})$ under the map

$$\lambda t \cdot TE_A(t, \vec{x})\colon\ ProgTerm(\vec{v}) \longrightarrow <\vec{x}> \subseteq A\,.$$

Therefore, an enumeration with repetitions of $<\vec{x}>$ is defined by the composition

$$te_A(j, \vec{x}) = TE_A(\gamma(j), \vec{x})$$

and the following diagram commutes:



and $te_A(j, \vec{x})$ is the value of the j-th term applied to $\vec{x}$. We will prove that te_A is a cov inductively definable enumeration.

4.7.6 Cov inductive definability of simple term evaluation

This theorem is the uniform version of Lemma 4.4.5 and is analogous to Theorem 1 in 3.6.4:

BASIC THEOREM. *Corresponding with the $r+2$ sorts $\vec{s} = (1, \ldots, r, \mathsf{N}, \mathsf{B})$ of Σ, there is an $r+2$ tuple $\vec{\alpha}$ of cov induction schemes of type $(M_1+1; \mathsf{N}, \vec{k}, \vec{s})$ such that for each $A \in \mathbb{K}$ and all $t \in ProgTerm(\vec{v})$ and $\vec{x} \in A[\vec{k}]$,*

$$\{\vec{\alpha}\}^A(\ulcorner t \urcorner, \vec{x}) = te_A(\ulcorner t \urcorner, \vec{x})\,.$$

PROOF. We prove the theorem by constructing the map te_A on A by induction on the complexity of the code $\ulcorner t \urcorner$. Let te_A^i be the i-th sorted component map of te_A, so:

$$te_A^i\colon\ \mathbb{N} \times R[\vec{v}, A] \longrightarrow A_i\,.$$

We define:

$$te_A^i(\ulcorner v_j \urcorner, \vec{x}) = x_j \qquad \text{if } v_j \text{ is of sort } i$$

$$te_A^i(\ulcorner F(t_1, \ldots, t_m) \urcorner, \vec{x}) = F^A(te_A^{i_1}(\ulcorner t_1 \urcorner, \vec{x}), \ldots, te_A^{i_m}(\ulcorner t_m \urcorner, \vec{x}))$$
$$\text{if } F \text{ is of type } (m; i_1, \ldots, i_m, i)$$

$$te_A^i(\ulcorner \textsf{if } b \textsf{ then } t_1 \textsf{ else } t_2 \textsf{ fi} \urcorner, \vec{x}) = \begin{cases} te_A^i(\ulcorner t_1 \urcorner, \vec{x}) & \text{if } te_A^{\mathbb{B}}(\ulcorner b \urcorner, \vec{x}) = \mathbb{t} \\ te_A^i(\ulcorner t_2 \urcorner, \vec{x}) & \text{if } te_A^{\mathbb{B}}(\ulcorner b \urcorner, \vec{x}) = \mathbb{f} \end{cases}$$

Thus (as seen before in 3.6.4) we are faced with a simultaneous course-of-values recursion involving a definition by cases. The case distinction is based upon the primitive recursive conditions that discover the status of $z \in \mathbb{N}$ as a code for a term; of which there are four, counting the default case that z is not a code for a term. (Notice that the case distinction on booleans, in the last clause, is an instance of the basic operation of definition by cases.) The degree of simultaneity is $r+2$ and the relevant

$$\delta_1(z), \ldots, \delta_r(z), \delta_{\mathbb{N}}(z), \delta_{\mathbb{B}}(z)$$

primitive recursively calculate codes for the subterms of z of the different sorts, where they exist; the degree of cov induction is essentially the maximum arity of the function symbols of Σ. By Corollary 2 in 4.5.5, there exist cov induction schemes to define te_A uniformly over all $A \in \mathbb{K}$. □

This result is the key to the proof: for our simulation of $T_A^*(S)$ by means of $G_A(S)$ on A is essentially a matter of primitive recursive computations on numerical codes for syntax supplemented by calls to term evaluation. This point will be prominent in the closing sections of this chapter.

4.7.7 The local representation map $G_A(S)$

We now specify the localized representation of $T_A^*(S)$, which consists of the maps Γ and $G_A(S)$, roughly described in 4.7.4.

From the enumeration of elements of A by terms

$$te_A\colon \mathbb{N} \times \boldsymbol{R}[\vec{v}, A] \longrightarrow A$$

we make an enumeration of the finite functions on A. The idea is that a finite function $\xi_i \in <\vec{x}>_{l_i}^*$ of the form $\xi_i\colon \mathbb{N} \longrightarrow <\vec{x}>_{l_i}^{\mu}$ may be represented by a code z for a finite sequence

$$z = \ulcorner(<n_1, z_1>, \ldots, <n_\lambda, z_\lambda>)\urcorner$$

wherein the n_α are all distinct, and a pair $<n_\alpha, z_\alpha>$ represents that the

element $te_A^{l_i}(z_\alpha, \vec{x}) \in <\vec{x}> \subseteq A$ is assigned as the value $\xi_i(n_\alpha)$; this, in turn, means that $te_A^{l_i}(z_\alpha, \vec{x})$ is assigned to the n_α-th location in the array a_i. Arguments of the function, and locations in the array, that do not appear in the sequence are assumed to be unspecified.

If z is the code for the sequence above then let $\boldsymbol{dom}(z) = \{n_1, \ldots, n_\lambda\}$.

We define the local enumeration of finite functions as a map

$$te_A^*: \mathbb{N} \times \boldsymbol{R}[\vec{v}, A] \times \mathbb{N} \longrightarrow A^{\mathbb{u}}$$

for which the i-th sorted component map

$$te_A^{*,i}: \mathbb{N} \times \boldsymbol{R}[\vec{v}, A] \times \mathbb{N} \longrightarrow A_i^{\mathbb{u}}$$

is defined by

$$te_A^{*,i}(z, \vec{x}, j) = \begin{cases} te_A^i(z_\alpha, \vec{x}) & \text{if } j \in \boldsymbol{dom}(z) \text{ and } j = n_\alpha \\ \mathbb{u} & \text{otherwise} \end{cases}$$

We prefer to write $te_A^{*,i}(z, \vec{x})(j)$ for $te_A^{*,i}(z, \vec{x}, j)$.

LEMMA. *Corresponding with the $r+2$ sorts $\vec{s} = (1, \ldots, r, \mathsf{N}, \mathsf{B})$ of Σ, there is an $r+2$ tuple $\vec{\beta}$ of cov induction schemes of type $(r+2; \mathsf{N}, \vec{k}, \mathsf{N}, \vec{s})$ over $\Sigma^{\mathbb{u}}$ such that for each $A \in \mathbb{K}$ and all $\vec{x} \in A[\vec{k}]$, $z, j \in \mathbb{N}$*

$$\{\vec{\beta}\}^{A^{\mathbb{u}}}(z, \vec{x}, j) = te_A^*(z, \vec{x})(j).$$

PROOF. Obvious, given 4.7.6. □

We can now construct the local coding

$$\Gamma: A[\vec{k}] \times \mathbb{N}^{M_1} \times \mathbb{N}^{M_2} \longrightarrow \boldsymbol{R}[\vec{v}, \vec{a}, A]$$

by simply defining, for $\vec{x} \in A[\vec{k}]$, and appropriate code vectors $z = (z_1, \ldots, z_{M_1})$ and $z' = (z'_1, \ldots, z'_{M_2})$, wherein

$$z'_i = \ulcorner (<n_{i1}, z_{i1}>, \ldots, <n_{i\lambda_i}, z_{i\lambda_i}>) \urcorner,$$

the value

$$\Gamma(\vec{x}, z, z') = (te_A(z_1, \vec{x}), \ldots, te_A(z_{M_1}, \vec{x}), te_A^*(z'_1, \vec{x}), \ldots, te_A^*(z'_{M_2}, \vec{x})).$$

Notice that, for notational simplicity, we continue to suppress many properties of the numerical arguments considered as codes for syntax. For example, no reference is made to the fact that each of the copies of $\mathbb{N}$ in the domain of Γ is the set of codes for elements, or finite functions of a particular sort. Nor do we give default values for our maps on numbers that do not qualify as appropriate codes.

Also, we must extend this Γ to embrace the error vector $\vec{\varepsilon}$ adjoined to $\boldsymbol{R}[\vec{v},\vec{a},A]$. Thus, we adjoin a special element $\ulcorner e \urcorner$ to the numerical domain $\mathbb{N}$ of A to make a "numerical" vector

$$e = (\ulcorner e \urcorner, \ldots, \ulcorner e \urcorner)$$

to code $\vec{\varepsilon}$ by defining

$$\Gamma_e\colon\ A[\vec{k}]\times\mathbb{N}^{M_1}\times\mathbb{N}^{M_2} \cup A[\vec{k}]\times\{e\} \to \boldsymbol{R}[\vec{v},\vec{a},A]\cup\{\vec{\varepsilon}\}$$

by setting $\Gamma_e(\vec{x},e)=\vec{\varepsilon}$ for all $\vec{x}\in A[\vec{k}]$. The adjunction of $\ulcorner e \urcorner$ to $\mathbb{N}$, and hence to A, we treat as essentially a notational device, accommodated by routine coding arguments on $\mathbb{N}$.

The codings Γ and Γ_e are clearly surjections (although they are not necessarily bijections).

The map $\boldsymbol{G}_A(S)$ is chosen to be a suitable map which commutes the following diagram:

$$\begin{array}{ccc} \boldsymbol{R}[\vec{v},\vec{a},A] & \xrightarrow{T_A^*(S)} & \boldsymbol{R}[\vec{v},\vec{a},A]\cup\{\vec{\varepsilon}\} \\ \Gamma \uparrow & & \uparrow \Gamma_e \\ A[\vec{k}]\times\mathbb{N}^{M_1}\times\mathbb{N}^{M_2} & \xrightarrow{\boldsymbol{G}_A(S)} & A[\vec{k}]\times\mathbb{N}^{M_1}\times\mathbb{N}^{M_2} \cup A[\vec{k}]\times\{e\} \end{array}$$

We partition the coordinate functions of $\boldsymbol{G}_A(S)$ thus:

$$\begin{aligned} &\boldsymbol{G}_0(S)\colon\ A[\vec{k}]\times\mathbb{N}^{M_1}\times\mathbb{N}^{M_2} \dashrightarrow A[\vec{k}] \\ &\boldsymbol{G}_{v_i}(S)\colon\ A[\vec{k}]\times\mathbb{N}^{M_1}\times\mathbb{N}^{M_2} \dashrightarrow \mathbb{N}\cup\{\ulcorner e \urcorner\} \qquad i=1,\ldots,M_1 \\ &\boldsymbol{G}_{a_i}(S)\colon\ A[\vec{k}]\times\mathbb{N}^{M_1}\times\mathbb{N}^{M_2} \dashrightarrow \mathbb{N}\cup\{\ulcorner e \urcorner\} \qquad i=1,\ldots,M_2 \end{aligned}$$

omitting the reference to A.

Observe that the key maps are the $\boldsymbol{G}_{v_i}(S)$ which calculate the code $\boldsymbol{G}_{v_i}(S)(\vec{x},z,z')$ of the value of the v_i on executing S on an input state whose values of $\vec{v}$ and $\vec{a}$ have codes z and z' respectively; the $\boldsymbol{G}_{a_i}(S)$ act similarly for the a_i. The map $\boldsymbol{G}_0(S)$ we take to be the projection

$$\boldsymbol{G}_0(S)(\vec{x},z,z') = \vec{x}\,.$$

Observe, too, that using $\ulcorner e \urcorner$ as a notational device for a numerical code, means that $G_A(S)$ is a map on A.

On proving that $G_A(S)$ is cov inductively definable, uniformly over A for $A \in \mathbb{K}$, the Basic Lemma of 4.7.3 may be deduced as follows: the required coordinate functions are definable as

$$T_{v_i}(S)(\vec{x}) \simeq te_A(G_{v_i}(S)(\vec{x}, \ulcorner \vec{v} \urcorner, \mathbf{null}), \vec{x})$$

$$T_{a_i}(S)(\vec{x})(j) \simeq te_A^*(G_{a_i}(S)(\vec{x}, \ulcorner \vec{v} \urcorner, \mathbf{null}), \vec{x})(j)$$

wherein $\ulcorner \vec{v} \urcorner$ codes the variables $\vec{v}$ and ***null*** codes a list of everywhere unspecified array variables $\vec{a}$. Notice that in this decomposition of the functions, the algebraic unspecified element ω appears only through the application of te_A^*: this explains the use of A^{ω} in the Basic Lemma in 4.7.3.

On proving that $G_A(S)$ is cov inductively definable, one appeals to the Basic Theorem in 4.7.6, and the Lemma earlier in this subsection, to conclude that the coordinate functions are cov inductively definable.

4.7.8 The map $G_A(S)$ is cov inductively definable

Thus, it remains to prove the following result:

LEMMA. *There is a $2M_1+M_2$ tuple $\vec{\alpha}$ of cov induction schemes such that for all $A \in \mathbb{K}$, there is a choice of $G_A(S)$ such that for all $\vec{x} \in A[\vec{k}]$, $(z, z') \in \mathbb{N}^{M_1} \times \mathbb{N}^{M_2}$*

$$\{\vec{\alpha}\}^A(\vec{x}, z, z') \simeq G_A(S)(\vec{x}, z, z')$$

Moreover, one can effectively calculate $\vec{\alpha}$ from S.

PROOF. At the heart of the proof is the construction of $G_A(S)$, by induction on the complexity of S, on A. From the construction, cov schemes $\vec{\alpha}$ will be seen to exist to define $G_A(S)$, uniformly over $A \in \mathbb{K}$; furthermore, it will be seen that the schemes are effectively calculable from S. Before starting the construction of $G_A(S)$ we must build machinery for evaluating program terms on the local representation space $A[\vec{k}] \times \mathbb{N}^{M_1} \times \mathbb{N}^{M_2}$.

4.7.9 Program term evaluation

Recall from 1.2.5 that the semantics $\mathcal{R}_A$ of program terms ***ProgTerm*** on PR.STATE(A) is defined as the union, over all sorts i, of maps

$$\mathcal{R}_A^i : \mathbf{ProgTerm}_i \to (\text{PR.STATE}(A) \to A_i^{\omega}).$$

Let $\mathbf{ProgTerm}(\vec{v}, \vec{a})$ be the collection of all program terms involving variables in $\vec{v}, \vec{a}$ only, and let $\mathbf{ProgTerm}_i(\vec{v}, \vec{a})$ be the subset of terms of sort i. Following the pattern of simple term evaluation in 4.7.5, we represent

$\mathcal{R}_A$ on $\boldsymbol{R}[\vec{v},\vec{a},A]$ by the *program term evaluation* map $\boldsymbol{PTE}_A$, which is the union over all sorts i of maps

$$\boldsymbol{PTE}_A^i:\ \boldsymbol{ProgTerm}_i(\vec{v},\vec{a})\times\boldsymbol{R}[\vec{v},\vec{a},A]\longrightarrow A_i^{\mathsf{u}}.$$

The i-th sort evaluation function $\boldsymbol{PTE}_A^i$ can be inductively defined on the structure of terms in the obvious way:

$$\boldsymbol{PTE}_A^i(v_j,\vec{x},\vec{\xi})=x_j \qquad \text{if } v_j \text{ is of sort } i$$

$$\boldsymbol{PTE}_A^i(a_j[t^{\mathsf{N}}],\vec{x},\vec{\xi})=\mathsf{Ap}_i^A(\xi_j,\boldsymbol{PTE}_A^{\mathsf{N}}(t^{\mathsf{N}},\vec{x},\vec{\xi}))$$
$$\text{if } a_j \text{ is of sort } i$$

$$\boldsymbol{PTE}_A^i(F(t_1,\ldots,t_m),\vec{x},\vec{\xi})=F^{A,\mathsf{u}}(\boldsymbol{PTE}_A^{i_1}(t_1,\vec{x},\vec{\xi}),\ldots$$
$$\ldots,\boldsymbol{PTE}_A^{i_m}(t_m,\vec{x},\vec{\xi}))$$
$$\text{where } F \text{ is of type } (m;i_1,\ldots,i_m,i)$$

$$\boldsymbol{PTE}_A^i(\textsf{if } b \textsf{ then } t_1 \textsf{ else } t_2 \textsf{ fi},\vec{x},\vec{\xi})=\begin{cases}\boldsymbol{PTE}_A^i(t_1,\vec{x},\vec{\xi}) & \text{if } \boldsymbol{PTE}_A^{\mathsf{B}}(b,\vec{x},\vec{\xi})=\mathsf{tt}\\ \boldsymbol{PTE}_A^i(t_2,\vec{x},\vec{\xi}) & \text{if } \boldsymbol{PTE}_A^{\mathsf{B}}(b,\vec{x},\vec{\xi})=\mathsf{ff}\\ \mathsf{u} & \text{if } \boldsymbol{PTE}_A^{\mathsf{B}}(b,\vec{x},\vec{\xi})=\mathsf{u}\end{cases}$$

(Recall the original definition of program term evaluation in 1.2.5, and the structure A^* in 2.3.1.)

4.7.10 Cov inductive definability of program term evaluation

In the construction of $G_A(S)$, a numerical representation $\boldsymbol{pte}_A$ of $\boldsymbol{PTE}_A$ is required that evaluates codes for program terms on the space $A[\vec{k}]\times\mathbb{N}^{M_1}\times\mathbb{N}^{M_2}$. To each $\boldsymbol{PTE}_A^i$ we associate a map

$$\boldsymbol{pte}_A^i:\mathbb{N}\times A[\vec{k}]\times\mathbb{N}^{M_1}\times\mathbb{N}^{M_2}\longrightarrow\mathbb{N}\cup\{\ulcorner\mathsf{u}\urcorner\}$$

such that for each $\vec{x}\in A[\vec{k}]$, $(z,z')\in\mathbb{N}^{M_1}\times\mathbb{N}^{M_2}$ and $t\in\boldsymbol{ProgTerm}_i(\vec{v},\vec{a})$ we have

$$\boldsymbol{PTE}_A^i(t,\Gamma(\vec{x},z,z'))=\mathsf{u}\ \text{ iff }\ \boldsymbol{pte}_A^i(\ulcorner t\urcorner,\vec{x},z,z')=\ulcorner\mathsf{u}\urcorner$$

and if $\boldsymbol{pte}_A^i(\ulcorner t\urcorner,\vec{x},z,z')\neq\ulcorner\mathsf{u}\urcorner$ then

$$\boldsymbol{PTE}_A^i(t,\Gamma(\vec{x},z,z'))=\boldsymbol{te}_A^i(\boldsymbol{pte}_A^i(\ulcorner t\urcorner,\vec{x},z,z'),\vec{x}).$$

Thus, as indicated, a special element $\ulcorner\mathsf{u}\urcorner$ is coded within the numerical domain of A to play the role of the unspecified algebraic element u in the numerical simulation. Again, as with $\ulcorner e\urcorner$, $\ulcorner\mathsf{u}\urcorner$ will be treated as a notational device and our discussion will remain focussed on A.

The construction of $\boldsymbol{pte}_A$, and the proof that it is cov inductively definable, are assisted by introducing certain auxiliary functions to do with A^*:

Let $\mathsf{Ap}_i^A : A_i^* \times \mathbb{N}^{\mathsf{u}} \to A_i^{\mathsf{u}}$ be locally represented by a map

$$\boldsymbol{ap}_i^A : \; A[\vec{k}] \times \mathbb{N} \times (\mathbb{N} \cup \{\ulcorner \mathsf{u} \urcorner\}) \to \mathbb{N} \cup \{\ulcorner \mathsf{u} \urcorner\}$$

in the following way: for $z = \ulcorner(\langle n_1, z_1\rangle, \ldots, \langle n_\lambda, z_\lambda\rangle)\urcorner$

$$\boldsymbol{ap}_i^A(\vec{x}, z, j) = \begin{cases} z_\alpha & \text{if } j \in \mathbb{N},\ \boldsymbol{te}_A^{\mathsf{N}}(j, \vec{x}) = n_\alpha \\ \ulcorner \mathsf{u} \urcorner & \text{if } j \in \mathbb{N},\ \boldsymbol{te}_A^{\mathsf{N}}(j, \vec{x}) \notin \boldsymbol{dom}(z) \\ \ulcorner \mathsf{u} \urcorner & \text{if } j = \ulcorner \mathsf{u} \urcorner. \end{cases}$$

Let $F^{A,\mathsf{u}} : A_{i_1}^{\mathsf{u}} \times \cdots \times A_{i_m}^{\mathsf{u}} \to A_i^{\mathsf{u}}$ be locally represented by some map

$$f : \; (\mathbb{N} \cup \{\ulcorner \mathsf{u} \urcorner\})^m \to \mathbb{N} \cup \{\ulcorner \mathsf{u} \urcorner\}$$

which satisfies the following:

$$f(j_1, \ldots, j_m) = \ulcorner \mathsf{u} \urcorner \text{ iff } \; j_\alpha = \ulcorner \mathsf{u} \urcorner \text{ for some } \alpha = 1, \ldots, m$$

and if $j_1, \ldots, j_m \neq \ulcorner \mathsf{u} \urcorner$ then

$$F^A(\boldsymbol{te}_A(j_1, \vec{x}), \ldots, \boldsymbol{te}_A(j_m, \vec{x})) = \boldsymbol{te}_A(f(j_1, \ldots, j_m), \vec{x}).$$

This last equation suggests we choose as f that map which tracks the application of the function symbol F to terms in $\boldsymbol{ProgTerm}(\vec{v})$, *i.e.*

$$\ulcorner F(t_1, \ldots, t_m)\urcorner = f(\ulcorner t_1 \urcorner, \ldots, \ulcorner t_m \urcorner)$$

and modify it to satisfy the first condition.

We now define $\boldsymbol{pte}_A^i$ in the obvious way, following the definition of $\boldsymbol{PTE}_A^i$:

$$\boldsymbol{pte}_A^i(\ulcorner v_j \urcorner, \vec{x}, z, z') = \ulcorner z_j \urcorner \qquad \text{if } v_j \text{ is of sort } i$$

$$\boldsymbol{pte}_A^i(\ulcorner a_j[t^{\mathsf{N}}] \urcorner, \vec{x}, z, z') = \boldsymbol{ap}_i^A(\vec{x}, z_j', \boldsymbol{pte}_A^{\mathsf{N}}(\ulcorner t^{\mathsf{N}} \urcorner, \vec{x}, z, z'))$$

$$\text{if } a_j \text{ is of sort } i$$

$$\boldsymbol{pte}_A^i(\ulcorner F(t_1, \ldots, t_m)\urcorner, \vec{x}, z, z') = f(\boldsymbol{pte}_A^{i_1}(\ulcorner t_1 \urcorner, \vec{x}, z, z'), \ldots$$
$$\ldots, \boldsymbol{pte}_A^{i_m}(\ulcorner t_m \urcorner, \vec{x}, z, z'))$$

$$\text{where } F \text{ is of type } (m; i_1, \ldots, i_m, i)$$

$$pte_A^i(\ulcorner \text{if } b \text{ then } t_1 \text{ else } t_2 \text{ fi}\urcorner, \vec{x}, z, z') =$$

$$\begin{cases} pte_A^i(\ulcorner t_1\urcorner, \vec{x}, z, z') & \text{if } pte_A^i(\ulcorner b\urcorner, \vec{x}, z, z') = y, \\ & \quad y \neq \ulcorner \mathbb{U}\urcorner \text{ and } te_A^{\mathbb{B}}(y, \vec{x}) = \mathbb{t} \\ pte_A^i(\ulcorner t_2\urcorner, \vec{x}, z, z') & \text{if } pte_A^i(\ulcorner b\urcorner, \vec{x}, z, z') = y, \\ & \quad y \neq \ulcorner \mathbb{U}\urcorner \text{ and } te_A^{\mathbb{B}}(y, \vec{x}) = \mathbb{f} \\ \ulcorner \mathbb{U}\urcorner & \text{if } pte_A^i(\ulcorner b\urcorner, \vec{x}, z, z') = \ulcorner \mathbb{U}\urcorner. \end{cases}$$

And we also define pte_A^i on the code for the error state:

$$pte_A^i(j, \vec{x}, e) = \ulcorner \mathbb{U}\urcorner$$

for all $\vec{x} \in A[\vec{k}]$.

4.7.11 Construction of $G_A(S)$

We construct $G_A(S)$ by induction on the complexity of S.

Basis. There are two cases.

Simple variable assignment case, $S \equiv v_i := t$. For $A \in \mathbb{K}$ and $\vec{x} \in A[\vec{k}]$, $\zeta = (z, z') \in \mathbb{N}^{M_1} \times \mathbb{N}^{M_2}$, we define $G_A(S)$ in two stages corresponding with the normal and error cases:

normal case: if $pte_A(\ulcorner t\urcorner, \vec{x}, \zeta) \neq \ulcorner \mathbb{U}\urcorner$ then

$$G_{v_i}(S)(\vec{x}, \zeta) = pte_A(\ulcorner t\urcorner, \vec{x}, \zeta)$$

$$G_{v_j}(S)(\vec{x}, \zeta) = z_j \quad \text{for } j = 1, \ldots, i-1, i+1, \ldots, M_1$$

$$G_{a_j}(S)(\vec{x}, \zeta) = z'_j \quad \text{for } j = 1, \ldots, M_2$$

error case: otherwise, if $pte_A(\ulcorner t\urcorner, \vec{x}, \zeta) = \ulcorner \mathbb{U}\urcorner$ then

$$G_{v_j}(S)(\vec{x}, \zeta) = \ulcorner e\urcorner \quad \text{for } j = 1, \ldots, M_1$$

$$G_{a_j}(S)(\vec{x}, \zeta) = \ulcorner e\urcorner \quad \text{for } j = 1, \ldots, M_2.$$

These coordinate functions can be defined by cov induction schemes $\vec{\alpha}$, using program term evaluation schemes and definition by cases.

Array variable assignment case, $S \equiv a_i[t^{\mathbb{N}}] := t$. For $A \in \mathbb{K}$ and $\vec{x} \in A[\vec{k}]$, $\zeta = (z, z') \in \mathbb{N}^{M_1} \times \mathbb{N}^{M_2}$, we define $G_A(S)$ in two stages:

normal case: if $\mathbf{pte}_A(\ulcorner t^{\mathbb{N}}\urcorner, \vec{x}, \zeta) \neq \ulcorner \mathbb{u} \urcorner$ and $\mathbf{pte}_A(\ulcorner t \urcorner, \vec{x}, \zeta) \neq \ulcorner \mathbb{u} \urcorner$ then let $y = \mathbf{te}_A(\mathbf{pte}_A(\ulcorner t^{\mathbb{N}}\urcorner, \vec{x}, \zeta), \vec{x})$ and

$$G_{a_i}(S)(\vec{x}, \zeta) = \begin{cases} \ulcorner(<n_{i1}, z_{i1}>, \ldots, <y, \mathbf{pte}_A(\ulcorner t \urcorner, \vec{x}, \zeta)>, \ldots \\ \qquad \ldots, <n_{i\lambda_i}, z_{i\lambda_i}>)\urcorner & \text{if } y \in \mathbf{dom}(z_i') \\ \ulcorner(z_i', <y, \mathbf{pte}_A(\ulcorner t \urcorner, \vec{x}, \zeta)>)\urcorner & \text{if } y \notin \mathbf{dom}(z_i') \end{cases}$$

$$G_{a_j}(S)(\vec{x}, \zeta) = z_j' \qquad \text{for } j = 1, \ldots, i-1, i+1, \ldots, M_2$$

$$G_{v_j}(S)(\vec{x}, \zeta) = z_j \qquad \text{for } j = 1, \ldots, M_1$$

error case: otherwise,

$$G_{v_j}(S)(\vec{x}, \zeta) = \ulcorner e \urcorner \qquad \text{for } j = 1, \ldots, M_1$$

$$G_{a_j}(S)(\vec{x}, \zeta) = \ulcorner e \urcorner \qquad \text{for } j = 1, \ldots, M_2.$$

These coordinate functions can be defined by cov induction schemes $\vec{\alpha}$, using program term evaluation schemes and definition by cases.

Induction step. There are three cases:

Composition case, $S \equiv S_1; S_2$. By the induction hypothesis there are $2M_1 + M_2$ tuples of schemes $\vec{\alpha}_1$ and $\vec{\alpha}_2$ to define the families of mappings

$$\{G_A(S_1) \mid A \in \mathbb{K}\} \quad \text{and} \quad \{G_A(S_2) \mid A \in \mathbb{K}\}$$

uniformly over $\mathbb{K}$. Clearly, for $A \in \mathbb{K}$, $\vec{x} \in A[\vec{k}]$, $\zeta \in \mathbb{N}^{M_1} \times \mathbb{N}^{M_2}$,

$$G_A(S)(\vec{x}, \zeta) \simeq \begin{cases} (\vec{x}, e) & \text{if } G_A(S_1)(\vec{x}, \zeta) \downarrow (\vec{x}, e) \\ (G_A(S_2) \circ G_A(S_1))(\vec{x}, \zeta) & \text{otherwise.} \end{cases}$$

Notice that the condition $G_A(S_1)(\vec{x}, \zeta) \downarrow (\vec{x}, e)$ depends ultimately upon the equality relation $\mathbf{eq}_{\mathbb{N}}: \mathbb{N} \times \mathbb{N} \to \mathbb{B}$. Thus, by means of general definition by cases, cov schemes $\vec{\alpha}$ can be made from $\vec{\alpha}_1$ and $\vec{\alpha}_2$ to define $\{G_A(S) \mid A \in \mathbb{K}\}$ uniformly over $\mathbb{K}$.

Conditional case, $S \equiv$ **if** b **then** S_1 **else** S_2 **fi**. By the induction hypothesis there are $2M_1 + M_2$ tuples of schemes $\vec{\alpha}_1$ and $\vec{\alpha}_2$ to define

$$\{G_A(S_1) \mid A \in \mathbb{K}\} \quad \text{and} \quad \{G_A(S_2) \mid A \in \mathbb{K}\}$$

uniformly over $\mathbb{K}$. For $A \in \mathbb{K}$, $\vec{x} \in A[\vec{k}]$ and $\zeta \in \mathbb{N}^{M_1} \times \mathbb{N}^{M_2}$ we define $G_A(S)$ in two stages:

normal case: if $\mathbf{pte}_A(\ulcorner b \urcorner, \vec{x}, \zeta) \neq \ulcorner \mathbb{u} \urcorner$ then

$$G_A(S)(\vec{x},\zeta) \simeq \begin{cases} G_A(S_1)(\vec{x},\zeta) & \text{if } \mathbf{te}_A(\mathbf{pte}_A(\ulcorner b \urcorner, \vec{x}, \zeta), \vec{x}) = \mathbb{t} \\ G_A(S_2)(\vec{x},\zeta) & \text{if } \mathbf{te}_A(\mathbf{pte}_A(\ulcorner b \urcorner, \vec{x}, \zeta), \vec{x}) = \mathbb{t} \end{cases}$$

error case: otherwise, if $\mathbf{pte}_A(\ulcorner b \urcorner, \vec{x}, \zeta) = \ulcorner \mathbb{u} \urcorner$ then

$$G_A(S)(\vec{x},\zeta) \simeq (\vec{x}, e).$$

By means of the cov schemes for simple term evaluation and program term evaluation, and definition by cases, cov induction schemes $\vec{\alpha}$ can be made from $\vec{\alpha}_1$ and $\vec{\alpha}_2$ to define $\{G_A(S) \mid A \in \mathbb{K}\}$.

Iteration case, $S \equiv$ **while** b **do** S_0 **od**. By the induction hypothesis, there is a $2M_1+M_2$ tuple of cov induction schemes $\vec{\alpha}_0$ to define $\{G_A(S_0) \mid A \in \mathbb{K}\}$ uniformly over $\mathbb{K}$. By means of the Iteration Lemma 4.5.6, one can construct a $2M_1+M_2$ tuple of cov induction schemes $\vec{\alpha}_I$ which defines, uniformly over $\mathbb{K}$, the following map corresponding with the bounded iteration of S_0: for $A \in \mathbb{K}$, $x \in A[\vec{k}]$ and $\zeta \in \mathbb{N}^{M_1} \times \mathbb{N}^{M_2}$

$$I_A(S_0)(0,\vec{x},\zeta) \simeq (\vec{x},\zeta)$$

$$I_A(S_0)(y+1,\vec{x},\zeta) \simeq \begin{cases} (\vec{x},e) & \text{if } I_A(S_0)(y,\vec{x},\zeta)\downarrow(\vec{x},e) \\ G_A(S_0) \circ I_A(S_0)(y,\vec{x},\zeta) & \text{otherwise.} \end{cases}$$

Now we consider the "length of loop" function

$$L_A(S): A[\vec{k}] \times \mathbb{N}^{M_1} \times \mathbb{N}^{M_2} \dashrightarrow \mathbb{N}$$

which counts the number of iterations of S_0 before the **'while'** construct is exited, either normally or exceptionally. This has the following definition:

$$\begin{aligned} L_A(S)(\vec{x},\zeta) \simeq (\mu y \in \mathbb{N})[\{ & \mathbf{pte}_A(\ulcorner b \urcorner, I_A(S_0)(y,\vec{x},\zeta)) = \ulcorner \mathbb{u} \urcorner \} \\ & \text{or } \{ \mathbf{pte}_A(\ulcorner b \urcorner, I_A(S_0)(y,\vec{x},\zeta))\downarrow \neq \ulcorner \mathbb{u} \urcorner \text{ and} \\ & \quad \mathbf{te}_A(\mathbf{pte}_A(\ulcorner b \urcorner, I_A(S_0)(y,\vec{x},\zeta)), \vec{x}) = \mathbb{f} \}]. \end{aligned}$$

Notice how program term evaluation in the definition detects errors caused by unspecified variables in b, and errors obtained earlier in the y-th execution of the body before the test b (because $\mathbf{pte}_A(i,\vec{x},e) = \ulcorner \mathbb{u} \urcorner$).

The family $\{L_A(S) \mid A \in \mathbb{K}\}$ is definable by a cov induction scheme α_L.

Finally, we can define

$$G_A(S)(\vec{x}, \zeta) \simeq I_A(S_0)(L_A(S)(\vec{x}, \zeta), \vec{x}, \zeta).$$

By means of general definition by cases, cov schemes $\vec{\alpha}$ can be made from α_I, α_L and the term evaluation schemes to define $\{G_A(S) \mid A \in \mathbb{K}\}$.

4.8 MORE ON INDUCTION

Primitive recursion and course-of-values recursion on $\mathbb{N}$ both define the primitive recursive functions on $\mathbb{N}$; indeed there are other interesting modifications of these recursions, allowing unbounded degrees of cov recursion and certain dependencies on parameters, which greatly increase the complexity of the computations generated, without leaving the class of primitive recursive functions: see Péter [1967], §5. Behind these equivalences is the primitive recursive coding of the natural numbers, of course.

Outside the classical case of the natural numbers, the two mechanisms of primitive recursion and course-of-values recursion are distinct in their power of definition. For 'while' programs are strictly less expressive than 'while'-array programs over standard structures. (This is commonly stated as: 'while' programs with counters are strictly less expressive than 'while'-array programs with counters.) Thus, the principal theorems of this chapter imply that the mechanisms are distinct in the presence of the least number operator. Yet they are inequivalent in the stronger sense that the primitive induction functions are a proper subclass of the primitive cov induction functions. The following is a very important observation:

THEOREM. *The term evaluation operator*

$$te_A : \mathbb{N} \times A[\vec{k}] \longrightarrow A$$

is not primitive inductively definable on every A, but it is primitive cov inductively definable on any A.

A proof of this can be made (albeit in a roundabout way) from results in Moldestad, Stoltenberg-Hansen and Tucker [1980a, 1980b] and Section 4.7. A direct proof will appear in a revision of Moldestad and Tucker [1981], which will include many results about the relationship between primitive recursion and simultaneous recursions of various degrees, generalized in an abstract setting.

The importance of term evaluation, as studied in 4.7.5 and 4.7.6, and used in the simulations of 4.7.7 and 4.7.11, is this:

Term evaluation is a tool that links computation on the abstract structure A to computation on syntax coded in $\mathbb{N}$. More precisely: *suppose a formalism for computation on A can be simulated by the partial recursive processing of its syntax coded in* $\mathbb{N}$, *together with calls to term evaluation to obtain semantic values in A. Then the formalism does not extend computationally that of cov inductive definability.*

To exemplify this in the case of formalisms for inductive definability, consider the following strengthening of the cov induction mechanism which introduces varying algebraic parameters in the recursion:

Course-of-values recursion with varying parameters.

Let $g_1, \ldots, g_m$ be functions with g_i of type $(n; \vec{k}, l_i)$ for $i = 1, \ldots, m$ and let $h_1, \ldots, h_m$ be functions with h_i of type $(n+1+dm; \mathsf{N}, \vec{k}, \vec{l}, \ldots, \vec{l}, l_i)$ (d times $\vec{l}$) for $i = 1, \ldots, m$. Let $\delta_1, \ldots, \delta_d$ be functions of type $(n+1; \mathsf{N}, \vec{k}, \mathsf{N})$ and let $\eta_1, \ldots, \eta_d$ be functions with η_j of type $(n+1; \mathsf{N}, \vec{k}, \vec{k})$ for $j = 1, \ldots, d$. Then define $f_1, \ldots, f_m$ by

$$f_i(0, \vec{x}) \simeq g_i(\vec{x})$$

and for $z > 0$,

$$f_i(z, \vec{x}) \simeq h_i(z, \vec{x}, f_1(\hat{\delta}_1(z, \vec{x}), \eta_1(z, \vec{x})), \ldots, f_m(\hat{\delta}_1(z, \vec{x}), \eta_1(z, \vec{x}))$$
$$\ldots, f_1(\hat{\delta}_d(z, \vec{x}), \eta_d(z, \vec{x})), \ldots, f_m(\hat{\delta}_d(z, \vec{x}), \eta_d(z, \vec{x})))$$

where, as before, $\hat{\delta}(z, \vec{x}) \simeq \min(\delta(z, \vec{x}), z \dot{-} 1)$.

The change in the computations generated by this new scheme is significant because the neat bottom-up structure of the computations, so useful in the proofs in Sections 4.3 and 4.6, has disappeared. But if cov recursion on $\mathbb{N}$ is exchanged for this cov recursion on A with variable parameters, then is a new class of functions obtained? The answer is No; because term evaluation is at hand to reduce the computations on A to computations on $\mathbb{N}$. (The reader is invited to reconsider the simulations of Section 4.7 in this way; and to show, more directly, that cov inductive definability with variable parameters implies '**while**'-array program computability, as their equivalence then follows.)

4.9 THE COV INDUCTIVELY DEFINABLE FUNCTIONS

The classes IND($\mathbb{K}$) and CIND($\mathbb{K}$) are both generalizations of the set of partial recursive functions on $\mathbb{N}$ to an arbitrary class $\mathbb{K}$ of standard many-sorted structures. The equivalence of cov inductive definability and 'while'-array program computability, and the significance of the cov inductive definability of term evaluation, suggests that cov inductive definability is a more useful, and more faithful, generalization for computability theory than inductive definability. This is true and, as we shall explain in the next section, the concept of cov inductive definability belongs at the centre of the theory of computation in an abstract setting; indeed the cov inductively definable functions can be made the subject of a *Generalized Church-Turing Thesis* of interest in applications to programming theory, logic and algebra (see Section 4.11). First, we pursue the comparison between the cov inductively definable functions and the partial recursive functions.

The theory of the partial recursive functions on $\mathbb{N}$ is based upon the *enumeration* or *indexing* of the set $\mathrm{PREC}(\mathbb{N})=\{\varphi_e \mid e\in\mathbb{N}\}$ of partial recursive functions, and basic results are the theorem about the existence of *universal partial recursive functions*, the *s-m-n Theorem*, and the various *recursion and fixed-point theorems* (see Kleene [1952], §§65, 66 or Rogers [1967], Chapters 1 and 11). The idea of indexing is equally important in the more general setting of $\mathbb{K}$:

The set $\boldsymbol{CIndSch}(\Sigma)$ of all cov induction schemes over the signature Σ can be enumerated by a primitive recursive Gödel numbering

$$\gamma\colon \mathbb{N} \longrightarrow \boldsymbol{CIndSch}(\Sigma)$$

and the code or index $e\in\mathbb{N}$ of a scheme $\gamma(e)$ transferred to the family

$$\mathcal{M}(\gamma(e)) = \{\mathcal{M}_A(\gamma(e)) \mid A\in\mathbb{K}\}$$

of partial functions it defines: recall 4.5.2. Thus, under this composition we get $\Gamma=\mathcal{M}\circ\gamma$ and write, after Kleene,

$$\Gamma(e) = \mathcal{M}(\gamma(e)) = \{e\}^{\mathbb{K}}$$

and, in particular, for $A\in\mathbb{K}$ we have $\Gamma=\{\Gamma_A \mid A\in\mathbb{K}\}$ where

$$\Gamma_A\colon \mathbb{N} \longrightarrow \mathrm{CIND}(A)$$

and

$$\Gamma_A(e) = \mathcal{M}_A(\gamma(e)) = \{e\}^A .$$

With this coding it possible to study the action of schemes on the class $\mathbb{K}$ using cov inductively definable functions:

UNIVERSALITY THEOREM. *Let* $\mathbb{K}$ *be a class. For each type* $(n;\vec{k},l)$ *there is a family*

$$\boldsymbol{U}[\vec{k},l] = \{\boldsymbol{U}[\vec{k},l]_A \mid A\in\mathbb{K}\}$$

of mappings of type $(n+1;\mathsf{N},\vec{k},l)$, *cov inductively definable uniformly over* $\mathbb{K}$, *such that for each* $A\in\mathbb{K}$, *the map*

$$\boldsymbol{U}[\vec{k},l]_A:\ \mathbb{N}\times A[\vec{k}] \dashrightarrow A_l$$

satisfies for all $e\in\mathbb{N}$, $\vec{x}\in A[\vec{k}]$,

$$\boldsymbol{U}[\vec{k},l]_A(e,\vec{x}) \simeq \{e\}^A(\vec{x}).$$

Moreover, an index $\boldsymbol{u}[\vec{k},l]$ *for the family* $\boldsymbol{U}[\vec{k},l]$ *is effectively calculable from the type* $(n;\vec{k},l)$.

The proof of the theorem is rather involved, and is founded on the now familiar idea of using term evaluation as a bridge between computation localized in A and computation on $\mathbb{N}$. Thus, the proof cannot be adapted to IND($\mathbb{K}$) and, indeed, there is no Universality Theorem for IND($\mathbb{K}$).

$\boldsymbol{S}$-m-$\vec{k}$ THEOREM. *Let* $\mathbb{K}$ *be a class. For each type* $(m+n;\mathsf{N},\ldots,\mathsf{N},\vec{k},l)$ (m *times* N), *there is a family*

$$\boldsymbol{S}[m,\vec{k}] = \{\boldsymbol{S}[m,\vec{k}]_A \mid A\in\mathbb{K}\}$$

of mappings of type $(m+1;\mathsf{N},\ldots,\mathsf{N},\mathsf{N})$ ($m+2$ *times* N), *cov inductively definable uniformly over* $\mathbb{K}$, *such that for each* $A\in\mathbb{K}$ *the map*

$$\boldsymbol{S}[m,\vec{k}]_A:\ \mathbb{N}\times\mathbb{N}^m \longrightarrow \mathbb{N}$$

satisfies for all $e\in\mathbb{N}$, $\vec{z}\in\mathbb{N}^m$, $\vec{x}\in A[\vec{k}]$,

$$\{\boldsymbol{S}[m,\vec{k}]_A(e,\vec{z})\}^A(\vec{x}) \simeq \{e\}^A(\vec{z},\vec{x}).$$

Moreover, an index $\boldsymbol{s}[m,\vec{k}]$ *for the family* $\boldsymbol{S}[m,\vec{k}]$ *is effectively calculable from the given type.*

These two theorems generalize, to the many-sorted case, results in Moldestad, Stoltenberg-Hansen and Tucker [1980b].

Many more results about the partial recursive functions (including the Recursion Theorems) may be generalized to the cov inductively definable functions on a class $\mathbb{K}$.

4.10 A SURVEY OF COMPUTABILITY IN AN ABSTRACT SETTING

The cov inductively definable functions can be distinguished as a class of functions definable by several disparate approaches to computability in an abstract setting. In this and the last section we will survey some of these approaches and discuss a Generalized Church-Turing Thesis that draws support from the equivalence theorems. The alternative methods for defining the cov inductively definable functions are to be found in various mathematical contexts and have various objectives; though, technically, they share the abstract setting of a *single-sorted abstract structure*. Nevertheless, here we suppose their common purpose to be the characterization of those functions effectively calculable in an abstract setting; and we assume their generalization to a class of many-sorted abstract structures as a matter of course.

In the context of the present work, the inductively definable and cov inductively definable functions are functions invented with the needs of logical representability and definability in mind.

Clearly, the various induction schemes on $\mathbb{N}$ are a primary source of technical ideas about *functions* computable on an abstract structure A. Although a generalization almost equivalent to the induction schemes of Section 4.1 was made early on, in Engeler [1968a], the methods have not been examined in detail or applied. (Let us note that inductive definability also has a pleasing theoretical and practical motivation in connection with parallel architectures: see Thompson and Tucker [1985].) These functions can also be characterized by approaches based upon

(*i*) machine models;
(*ii*) high-level programming constructs;
(*iii*) axiomatic methods;
(*iv*) equational calculi;
(*v*) fixed-point methods;
(*vi*) set theoretic methods.

We will say something about each in turn, concentrating on the first three approaches, as they are directly relevant to the concerns of the present work.

4.10.1 Machine models

Perhaps the most concrete approach to generalizing computability theory from $\mathbb{N}$ to an abstract structure A is that based upon models of machines that handle data from A. To be specific, we consider some models called

A-register machines that generalize to A the register machine models on $\mathbb{N}$ in Shepherdson and Sturgis [1963] (see also Cutland [1980] for a development of recursive function theory using register machines); the first A-register machines appeared in Friedman [1971].

The study of these machine models on A reveals some basic combinatorial properties of computation in an abstract setting that have no analogue in computation on $\mathbb{N}$. In order to explain these features we will discuss the models for *an arbitrary single-sorted relational structure* A.

An *A-register machine* has a fixed number of registers, each of which can hold a single element of A. The machine can perform the basic operations of A and decide the basic relations of A; in addition, it can relocate data and test when two registers carry the same element.

Thus, the programming language that defines the A-register machine has register names or variables $r_0, r_1, r_2, \ldots, r_n, \ldots$ and labels $0, 1, 2, \ldots$, and allows instructions of the form

$$r_\lambda := F(r_{\mu_1}, \ldots, r_{\mu_m})$$

$$r_\lambda := c$$

$$r_\lambda := r_\mu$$

$$\text{if } R(r_{\mu_1}, \ldots, r_{\mu_m}) \text{ then } i \text{ else } j$$

$$\text{if } r_\lambda = r_\mu \text{ then } i \text{ else } j$$

wherein $\lambda, \mu, \mu_1, \ldots, \mu_m \in \mathbb{N}$; $i, j \in \mathbb{N}$ and are considered as labels; and F, c, R are symbols for a basic operation, constant and relation respectively.

A program for an A-register machine is called a *finite algorithmic procedure* or *fap*, and it has the form of a finite numbered or labelled list of A-register machine instructions

$$I_1, \ldots, I_n .$$

Given a formal definition of a machine state, containing the contents of registers and the label of a current instruction, is it easy to give a formal semantics for the finite algorithmic procedures — one in which the instructions are given their conventional meaning.

On setting conventions for input and output registers we obtain the class FAP(A) of *all partial functions on A computable by all finite algorithmic procedures on A-register machines.*

Secondly, an *A-register machine with counting* is an A-register machine enhanced with a fixed, finite number of *counting registers.* Each counting register can hold a single element of $\mathbb{N}$ and the machine is able to put zero into a counting register, add or subtract one from a counting register, and test when two counting registers contain the same number. Thus, an

A-register machine with counting is an A-register machine augmented by a conventional register machine on $\mathbb{N}$.

The programming language that defines the A-register machine with counting has new variables $c_0, c_1, c_2, \ldots$ for counting registers and new instructions

$$c_\lambda := 0$$
$$c_\lambda := c_\mu + 1$$
$$c_\lambda := c_\mu \dot{-} 1$$
$$\text{if } c_\lambda = c_\mu \text{ then } i \text{ else } j$$

for $\lambda, \mu \in \mathbb{N}$ and $i, j \in \mathbb{N}$ considered as labels.

A program for an A-register machine with counting is called a *finite algorithmic procedure with counting*, or *fapC*, and is a finite numbered list of machine instructions. Once again it is easy to give a formal semantics for the language and to rigorously define the class FAPC(A) of *all partial functions on A computable by A-register machines with counting.*

Notice that this machine model is directly concerned with the process of "standardization" of the structure A by the addition of the natural numbers. However, the two-sorted structure obtained has as numerical domain *Presburger Arithmetic*,

$$(\mathbb{N};\ 0, x+1, x \dot{-} 1)$$

rather than the standard model of the natural numbers, which has multiplication. Computationally the structures are equivalent, but they are radically different in their program correctness theories (see Bergstra and Tucker [1982a]). *The point of this model is that it enhances computation on the abstract algebraic structure A with computation on* $\mathbb{N}$.

The A-register machine and A-register machine with counting, and their classes of partial functions FAP(A) and FAPC(A), were introduced in Friedman [1971].

Next, an *A-register machine with stacking* is an A-register machine augmented with a stacking device into which the entire contents of the algebraic registers of the A-register machine can be copied at various points in the course of a computation.

The programming language that defines the A-register machine with stacking has a new variable s for the store or stack and the new instructions:

$$\textsf{stack}(i, r_0, \ldots, r_m)$$
$$\textsf{restore}(r_0, \ldots, r_{i-1}, r_{i+1}, \ldots, r_m)$$
$$\textsf{if } s = \emptyset \textsf{ then } i \textsf{ else marker}.$$

Here $i \in \mathbb{N}$ is considered as a label. Intuitively, their meaning for the machine are as follows: the **'stack'** instruction commands the device to copy the contents of all the registers and store at the top of a (single) stack, along with the instruction label i. The **'restore'** instruction returns to the registers $r_0, \ldots, r_{i-1}, r_{i+1}, \ldots, r_m$ the values stored at the top of the stack; the value of r_i is lost (in order not to destroy the result of the sub-computation preceeding the **'restore'** instruction), as is the instruction label. The test instruction passes control to instruction i if the stack is empty and to the instruction indexed by the label contained in the top-most element of the stack otherwise.

A program for an A-register machine with stacking is called a *finite algorithmic procedure with stacking*, or *fapS*, and is a finite numbered list of machine instructions. On formalizing the semantics for the language, it is easy to define the class FAPS(A) of all partial functions on A computable by A-register machines with stacking.

Of course there are many alternative designs for a stacking device of equivalent computational power (we have chosen a mechanism that is generous in regard of its algebraic storage and control). *The point of this model is that it enhances the bounded finite algebraic memory available in computation by an A-register machine with unbounded finite algebraic storage.*

Finally, an *A-register machine with counting and stacking* is an A-register machine augmented by both a counting and stacking device. A program for such a machine is called a *finite algorithmic procedure with counting and stacking*, or *fapCS*, and the class of all partial functions on A computable by such machines is denoted FAPCS(A).

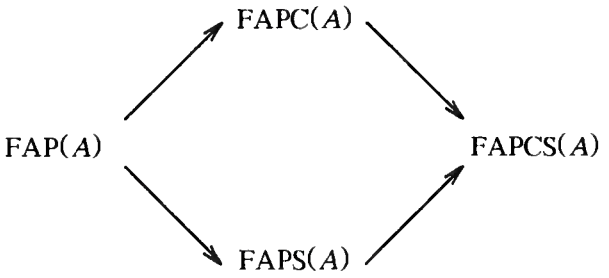
This stack device and its associated classes of functions FAPS(A) and FAPCS(A), were introduced in Moldestad, Stoltenberg-Hansen and Tucker [1980a, 1980b].

Now in the case of computability of the natural numbers $A = \mathbb{N}$ we have

$$\text{FAP}(\mathbb{N}) = \text{FAPC}(\mathbb{N}) = \text{FAPS}(\mathbb{N}) = \text{FAPCS}(\mathbb{N})$$

but in the abstract setting we have:

THEOREM. *For any single-sorted structure A, the relationship between the sets of functions is*



Moreover, there exists a structure on which the above inclusions are strict.

This theorem is taken from Moldestad, Stoltenberg-Hansen and Tucker [1980b]. It and other results about these models make clear the fact that, when computing in the abstract setting of a structure A, adding

computation on $\mathbb{N}$

unbounded algebraic memory over A

both separately, and together, increases the computational power of the formalism.

The connection with the inductive definability formalism is easily explained. Assuming the straight-forward generalization of the machine models to accommodate many-sorted structures, we have

THEOREM. *For any standard many-sorted structure A,*

$$\mathrm{IND}(A) = \mathrm{FAPC}(A)$$
$$\mathrm{CIND}(A) = \mathrm{FAPCS}(A).$$

Two other machine model formalisms of interest are the *A-register machines with index registers* and *countable algorithmic procedures* in Shepherdson [1975] and the *generalized Turing algorithms* in Friedman [1971]; these are equivalent to cov inductive definability; in the obvious notations:

THEOREM. *For any standard many-sorted structure A,*

$$\mathrm{FAPCS}(A) = \mathrm{GTA}(A) = \mathrm{FAPIR}(A) = \mathrm{CAP}(A) = \mathrm{CIND}(A).$$

In addition, it is convenient at this point to mention Friedman's *effective definitional schemes* which are a simple and transparent technical device for defining and analyzing computability on A. The effective definitional schemes have found a useful role in the logic of programs (see Tiuryn [1981], for example).

THEOREM. *For any standard many-sorted structure A,*

$$\mathrm{FAPCS}(A) = \mathrm{EDS}(A) = \mathrm{CIND}(A).$$

4.10.2 High-level programming constructs; program schemes

The study of computability via machine models is akin to low level programming, where there is a simple correspondence between instructions and machine operations. In high-level programming, abstractions away from the machine are achieved wherein a program statement or command can set off a sequence of machine operations. This break with programming a specific architecture increases the practical need for mathematical semantics.

We have, of course, already studied some high-level constructs in the languages for

(i) 'while'-array programs, and
(ii) recursion-array programs.

Both formalisms are equivalent with cov inductive definability. (See Sections 4.6 and 4.7 for a detailed account of (i); the argument for (ii) can then be presented as a "programming exercise", showing that (i) and (ii) are equivalent languages. A direct argument for (ii) requires a refinement of Section 3.6 using the techniques of Section 4.7.)

However, in contemplating high-level constructs with regard to generalizing computability theory, close attention must be paid to the special ideas about algorithms that motivate their introduction. Clearly, recursion and iteration are quite distinct tools for *defining and implementing algorithms*. But non-deterministic constructs, such as those in the *guarded command language* (Dijkstra [1975]), or the *non-deterministic assignment*

$$x := y.\Phi(x, y),$$

where Φ is some condition relating y to x (Back [1983]), or the *random assignment*

$$x := ?$$

(Apt and Plotkin [1986]), are constructs proposed as tools for *algorithm design and specification*, in order to abstract away from algorithmic implementation.

In these early stages of building a generalization, we think it prudent to concentrate on making a deterministic theory, having clear relations with "classical" computability theory on $\mathbb{N}$; and therefore to neglect these important specification constructs for the present. Technically, to appreciate such non-deterministic constructs, a deterministic theory is a necessary prerequisite. Unfortunately, there are many unanswered questions as to the nature of the relationships between non-determinism and specification, and determinism and implementation. The programming of computations involving concurrency and communication is also an important territory we here leave unexplored. (See Thompson and Tucker [1985], Thompson [1987] and Martin and Tucker [1987] for work that exploits the parallelism inherent in the simultaneous induction schemes used here.) We will return to the broad theme of programming languages and computability theory in Section 4.11.

Instead, we will briefly draw attention to a body of early work on the computational power of elementary control and data structures.

The classification of programming features such as iterations, recursions, 'goto's, arrays, stacks, queues, lists seems to have begun in earnest with Luckham, Park and Paterson [1970] and Paterson and Hewitt [1970]. The central notions are that of a *program scheme* and its *interpretation* on a *model*, and that of the *equivalence* of program schemes on *all* models. These ideas may be considered as direct technical precursors of the corresponding syntactic and semantic concepts we use here, namely: program, state transformer semantics, abstract data type, equivalence on $\mathbb{K}$. The importance of a general syntactic notion of a program scheme that can be applied to abstract structures was first discussed in Luckham and Park [1964] and in Engeler [1968b]. We note that in the latter paper computation over arbitrary classes of structures is treated in the course of analyzing program termination by means of logical formulae from a fragment of $L_{\omega_1,\omega}$; Engeler [1968b] is the origin of algorithmic and dynamic logic.

The study of the power of programming features came to be known as *program schematology*. Like program verification, the subject was contemporary with, but independent of, research on programming language semantics (recall Section 0.1). The necessity of introducing abstract structures in such a classification project is easy to understand. From the point of view of programming theory the equivalence of most algorithmic formalisms for computing on $\mathbb{N}$ with the partial recursive functions on $\mathbb{N}$ is a mixed blessing. This stability of the computational models illuminates our perception of the scope and limits of computer languages and architectures; and has many technical applications in the mathematical theory of computation (recall the origins of this chapter in Chapter 3). However,

the restriction to $\mathbb{N}$ fails to support an analysis of the intuitive differences between programming with and without arrays, 'goto's, boolean variables, and so forth.

The research on schematology has produced many program constructs that are weaker or equivalent with those four basic classes we have discussed. We refer the reader to the book Greibach [1975] for a general introduction to schematology and, in particular, to Shepherdson [1985] for a detailed discussion of many important results and their relation to machine models. Other significant references are Constable and Gries [1972], Chandra [1973], Chandra and Manna [1975].

4.10.3 Axiomatic methods

In an axiomatic method one defines the concept of a *computation theory* as a set $\Theta(A)$ of partial functions on a structure A having some of the essential properties of the set of partial recursive functions on $\mathbb{N}$. To take an example, $\Theta(A)$ can be required to contain the basic algebraic operators of A; be closed under operations such as *composition*; and, in particular, possess an enumeration for which appropriate *universality* and *s-m-n properties* are true. Thus, in Section 4.9, we explained that $\mathrm{CIND}(A)$ is a computation theory in this sense. We also showed implicitly how the idea uniformizes over $\mathbb{K}$: a computation theory over $\mathbb{K}$ is a class $\Theta(\mathbb{K})=\{\Theta(A) \mid A\in\mathbb{K}\}$ with a uniform enumeration of an appropriate kind.

It is important to note that computation theory definitions, of which there are a number of equivalent examples, require $\mathbb{N}$ to be part of the underlying structures A for the indexing of functions; here (current) *axiomatic methods specifically address standard structures and classes of standard structures.*

With reference to the definition sketched above, the following theorem is of importance here:

THEOREM. *The set of cov inductively definable functions* $\mathrm{CIND}(A)$ *on a structure* A *is the smallest set of partial functions on* A *to satisfy the axioms of a computation theory; in consequence,* $\mathrm{CIND}(A)$ *is a subset of every computational theory* $\Theta(A)$ *on* A.

The definition of a computation theory favored here is from Fenstad [1975, 1980] which takes up the initiative in Moschovakis [1971]. We note that *the cov inductively definable functions coincide with the prime computable functions of Moschovakis.* The above theorem can be deduced using work in Moldestad, Stoltenberg-Hansen and Tucker [1981b]; see also Chapter 0 in Fenstad [1980].

4.10.4 Equational definability

One of the earliest formalizations of effective computability was by means of *functions effectively reckonable in an equational calculus*, a method known as *equational* or *Herbrand-Gödel-Kleene definability*. This was the method employed to define the recursive functions in important writings such as Church [1936] and Kleene [1952].

Equational definability may be generalized from $\mathbb{N}$ to an arbitrary structure A with the natural result that, in case A is a standard structure, equational definability is equivalent with cov inductive definability. The first attempt at such a generalization is Lambert [1968]. However, we prefer to sketch a simpler treatment from Moldestad and Tucker [1981].

First we choose a language $\boldsymbol{EL} = \boldsymbol{EL}(\Sigma)$ for defining equations over signature Σ and transforming them in simple deductions. Let $\boldsymbol{EL}$ have constants $a, b, c, \ldots$ and variables $x, y, z, \ldots$ for data; and variables $p, q, r, \ldots$ for functions. Using the basic operations of the signature we inductively define terms in the usual way, namely:

$$t ::= a \mid x \mid p(t_1, \ldots, t_n) \mid F(t_1, \ldots, t_k) \mid \textsf{if } t^{\mathsf{B}} \textsf{ then } t_1 \textsf{ else } t_2 \textsf{ fi}.$$

An *equation* in $\boldsymbol{EL}$ is any expression $e \equiv (t_1 = t_2)$.

Let E be a set of equations. An equation e is defined to be *formally derivable or deducible* from the equations of E, written $E \vdash e$, if one of the following conditions holds:

(*i*) $e \in E$;

(*ii*) e is obtained from some e', such that $E \vdash e'$, by replacing every occurrence of a variable x in e' by a constant c;

(*iii*) e is obtained from some e', such that $E \vdash e'$, by replacing at least one occurrence of a subterm t of e' by a constant c, where t has no free variables and $E \vdash t = c$.

A deduction of an equation e from a set of equations E is a list $e_1, \ldots, e_k$ of equations such that $E \vdash e_i$ for all i and $e_k \equiv e$.

Thus, it remains to formulate equational deductions with respect to a given structure A of signature Σ in order to formulate what it means for a function f on A to be *equationally definable* on A. This is essentially giving our system a semantics. The first semantical problem is to allow the basic operations of A to play a role in deductions from a set of equations E and this is accomplished by permitting

$$E \vdash F(c_1, \ldots, c_n) = c \qquad \text{if } F^A(c_1^A, \ldots, c_n^A) = c^A$$
$$E \vdash \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi} = c_1 \quad \text{if } b^A = \mathbb{t}$$
$$E \vdash \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi} = c_2 \quad \text{if } b^A = \mathbb{f}$$

This is the reason we add the constants to ***EL***.

The second semantical problem is to prove a single-valuedness property of the form:

$$E \vdash p(c_1, \ldots, c_n) = a_1 \text{ and } E \vdash p(c_1, \ldots, c_n) = a_2 \implies a_1 = a_2.$$

This done, we can define $f: A[\vec{k}] \to A$ to be *equationally definable over* A if for some finite set of equations E and some function symbol p,

$$E \vdash p(c_1, \ldots, c_n) = c \text{ iff } f(c_1^A, \ldots, c_n^A) = c^A$$

for all constants of ***EL***.

Let EQN(A) denote the set of all equationally definable functions on A.

THEOREM. *For any structure* A

$$\text{EQN}(A) = \text{FAPS}(A)$$

and, in particular, for any standard structure A

$$\text{EQN}(A) = \text{CIND}(A).$$

4.10.5 Fixed-point methods

The familiar definition of the recursive functions on $\mathbb{N}$ based on the primitive recursion scheme of Dedekind and Gödel, and the least number operator of Kleene, appeared in Kleene [1936]. In Kleene [1959, 1963] appeared a thorough revision of the process of recursion on $\mathbb{N}$ sufficiently general to include recursion in objects of higher function type. In Platek [1966] there is an abstract account of higher-type recursion. The central technical notion is that of the explicit definition of fixed-point operators.

In Moldestad, Stoltenberg-Hansen and Tucker [1981a] Platek's methods were analysed and classified in terms of the machine models of 4.10.1. Like equational definability, definability by fixed-point operators applies to an arbitrary structure A and is there equivalent to fapS-computability. Thus, this notion coincides with cov-inductive definability of A in a standard structure. We will sketch the method.

First we construct the language $\boldsymbol{FL} = \boldsymbol{FL}(\Sigma)$ for defining fixed-point operators. Let ***FL*** have the data and function variables of ***EL***, the equation language of 4.10.4. Using the basic operations of the signature Σ and the λ-abstraction notation we create a set of *fixed-point terms* of both data

and function types:

$$t ::= x \mid p \mid F \mid \mathsf{DC} \mid T(t_1, \ldots, t_n) \mid \mathsf{FP}[\lambda p \cdot y_1, \ldots, y_n \cdot t].$$

Here p is a function variable, F is a basic operation of Σ, DC is the definition by cases functional (all of type function), T is a term of type function, $t_1, \ldots, t_n$ and t are terms of type data, and $y_1, \ldots, y_n$ are data variables.

Each term defines a function on each structure A of signature Σ. The definition of the semantics of terms is by induction on their construction, the terms of the form

$$\mathsf{FP}[\lambda p \cdot y_1, \ldots, y_n \cdot t]$$

being assigned the unique least fixed-point of the continuous monotonic operator defined by the notation $\lambda p \cdot y_1, \ldots, y_n . t$.

A function $f: A[\vec{k}] \to A$ is definable by fixed-point terms over A if there is a term t such that for all $\boldsymbol{x} \in A[\vec{k}]$, $f(\boldsymbol{x}) \simeq t(\boldsymbol{x})$.

Let FPD(A) denote the set of all functions definable by fixed-point terms over A.

THEOREM. *For any structure A,*

$$\mathrm{FPD}(A) = \mathrm{FAPS}(A)$$

and, in particular, for any standard structure A

$$\mathrm{FPD}(A) = \mathrm{CIND}(A).$$

For the many necessary details see Moldestad, Stoltenberg-Hansen and Tucker [1981a].

4.10.6 Set recursion

Given a structure on A one can construct a set-theoretic hierarchy H(A) over A, taking A as urelements, and, depending upon the construction, develop a recursion theory on H(A). This is the methodology in Normann [1978] where combinatorial operations on sets are employed to make a generalization of computability. In Moldestad and Tucker [1981], Normann's set recursion schemes are applied to the domain HF(A), the set of hereditarily finite subsets so as to invest the general construction with computational content. HF(A) is inductively defined as follows:

(i) $A \subseteq \mathrm{HF}(A)$

(ii) if $a_1, \ldots, a_n \in \mathrm{HF}(A)$ then $\{a_1, \ldots, a_n\} \in \mathrm{HF}(A)$, $n \geq 0$.

Thus, $\emptyset \in \mathrm{HF}(A)$, a copy of $\mathbb{N} \subset \mathrm{HF}(A)$ and copies of cartesian products of A are embedded in HF(A). From computability on HF(A) a notion of

computability on A may be easily obtained. It is the case that *a function on A is set-recursive if, and and only if, it is cov-inductively definable on A.*

4.10.7 Some other approaches

Finally, we will record for completeness some further notions of finite computability which are significant, but which do *not* coincide with cov-inductive definability.

In Moschovakis [1969] there are *three* classes of functions on a standard structure of interest. The *absolutely prime computable functions* on a standard structure A are precisely CIND(A). The *prime computable functions* on A are those functions definable by the cov induction schemes *enhanced by allowing all constant functions* on A.

The third class of functions is called the class of *search computable functions* and is made by adjoining to prime computability a global search operator on A, justified by the postulation of a well-ordering on A or by the axiom of choice. Such a search operator can also be brought into play in the definition of cov inductive definability to preserve the equivalence of the concepts. The technical value of the non-constructive search operator lies in its role in linking computability and logical definability on abstract structures: without search the class of semicomputable sets need not be closed under existential quantification.

In Montague [1968] semicomputability is identified with Σ_1-definability in a certain infinitary language. Taking Montague's cardinal index $\kappa = \aleph_0$ one obtains a finite definability which is proved, in Gordon [1970], to be equivalent to search computability. A similar situation obtains in the more general set-theoretic definability approach on HF(A) in Barwise [1975].

Thus, in working with the class of cov inductively definable functions we neglect the constant functions and Σ_1-definability in favour of a stricter interpretation of finite computability in an abstract setting.

4.11 A GENERALIZED CHURCH-TURING THESIS

The cov inductively definable functions are a mathematically interesting and useful generalization of the partial recursive functions on $\mathbb{N}$ to an abstract structure A and class $\mathbb{K}$ of structures. Do the cov inductively definable functions also give rise to an interesting and useful generalization to A and $\mathbb{K}$ of the Church-Turing Thesis, concerning effective computability on $\mathbb{N}$?

They do; though this answer is difficult to explain fully and impossible to explain briefly. In this last section we can only *sketch* some reasons directly relevant to the concerns of this monograph. Two subjects will receive attention: the idea of effective calculability in an abstract setting, and the status of a Church-Turing Thesis in programming language theory. We will formulate *two* Generalized Church-Turing Theses that we believe are consistent and useful for work on programming languages.

4.11.1 Some first attempts at generalizations

Consider the following statement:

A NAIVE GENERALIZED CHURCH-TURING THESIS. *The functions and relations effectively calculable on a many-sorted abstract structure A, or class* $\mathbb{K}$ *of structures, are the functions and relations cov inductively definable on A, or definable uniformly over* $\mathbb{K}$, *respectively.*

What can be meant by "effective calculability" on an abstract structure or class of structures?

The idea of effective calculability is complicated in the standard situation of calculation with $\mathbb{N}$, as it is made up from many philosophical and mathematical ideas about the nature of finite computation with finite or concrete elements. For example, its analysis raised questions about the mechanical representation and manipulation of finite symbols; and about the formalization of *constituent concepts* such as algorithm; deterministic procedure; computer program; functions definable by these entities; functions reckonable in formal systems; functions computable by machines; and so on.

The idea of effective calculability is particularly deep and valuable because of the close relationships that can be shown to exist between its distinct constituent concepts. Only *some* of these constituent concepts can be re-interpreted or generalized to work in an abstract setting; and hence the general concept (and the term) of effective calculability does not belong in a generalization of the Church-Turing Thesis. In addition, since finite computation on finite data is truly a fundamental phenomenon, it is approriate to preserve the term with its established meaning.

In seeking a generalization of the Church-Turing Thesis we are trying to make explicit certain primary informal concepts that are formalized by the technical definitions, and hence to clarify the nature and use of the computable functions in programming language theory.

We will start by trying to clarify the nature and use of abstract structures. Mathematically, there are three points of view from which to consider the step from concrete structures to abstract structures, and hence

three points of view from which to consider the cov inductively definable functions.

First, there is abstract algebra, which is a theory of calculation based upon the "behaviour" of elements in calculations without reference to their "nature". This abstraction is achieved through the concept of isomorphism between concrete structures; an abstract structure A is a concrete structure considered unique only up to isomorphism.

Secondly, there is logic and, in particular, model theory, which is a theory about the scope and limits of axiomatizations and proofs. Here structures and classes of structures are used to discuss formal systems in terms of consistency, soundness, completeness, and so on.

Thirdly, in programming language theory, there is data type theory, which is about data types that users may care to define and that arise independently of programming languages. Here structures are employed to discuss the semantical structure of data types, and isomorphisms are employed to make the semantics independent of implementations. In addition, axiomatic theories are employed to discuss their specifications and implementation.

Data type theory is mathematically dependent upon the first two subjects and is our main point of view.

Each of these theories, because of its special concerns and technical emphasis, leads to a distinct theory of computability on abstract structures.

Suppose, for example, the cov inductively definable functions are considered with the needs of doing algebra in mind. Then the context of studying algorithms and decision problems for groups, rings and fields, etc., leads to a formalization of a Generalized Church-Turing Thesis tailored to use by an algebraist:

GENERALIZED CHURCH-TURING THESIS FOR ALGEBRA. *The set of functions and relations on an abstract algebraic structure A, or class $\mathbb{K}$ of algebras, that are calculable by means of finite deterministic algebraic algorithms on A, or uniformly over the algebras of $\mathbb{K}$, is the set of functions and relations cov inductively definable on A, or uniformly over $\mathbb{K}$.*

A detailed account of computability on abstract structures from the point of view of algebra is Tucker [1980].

We will formulate two generalizations tailored to the uses of a programming language designer. The first is essentially a translation of the algebraist's thesis into the terminology of the theory of abstract data types (important: recall Section 0.3, if necessary):

GENERALIZED CHURCH-TURING THESIS FOR PROGRAMMING LANGUAGES: VERSION I. *Consider a deterministic programming language over an abstract data type* **dt**. *The set of functions and relations on a structure A, representing an implementation of the abstract type* **dt**, *that can be programmed in the language, is contained in the set* CIND(A) *of cov inductively definable functions and relations. The class of functions and relations over a class* $\mathbb{K}$ *of structures, representing a class of implementations of the abstract type* **dt**, *that can be programmed in the language, uniformly over all implementations of* $\mathbb{K}$, *is contained in the class* CIND($\mathbb{K}$).

The second generalization (given in 4.11.3) is a refinement of the first, obtained by considering the implementation of functions and relations in an arbitrary (not necessarily deterministic) programming language. In programming language theory, the computational equivalence of *different* languages is a central concern, and one in which computable functions and relations, and a generalized Church-Turing Thesis, will play a leading role. This we will now explain.

4.11.2 Equivalence of programming languages

In the theory of programming languages we are interested in properties of languages for expressing algorithms and their underlying models of computation. Thus, we are interested in properties of languages such as program transformations; program equivalence; specification methods; verification methods; side-effects; errors; run-time and space performance; and so on. What is the role of the classes of functions and relations computable by the languages?

The computable functions and relations allow us to compare *different* programming languages; in particular, they allow us to prove that certain disparate programming languages compute the same classes of functions and relations. Thus, classes of functions and relations are used to classify programming languages in terms of their computational power. In this classification the Church-Turing Thesis is often used (for example) to justify the criterion that *for a formalism to be a general purpose programming language it is necessary and sufficient that it can implement the partial recursive functions on* $\mathbb{N}$.

Let us reconsider the equivalence of programming languages, as described in Section 4.10 in the style conventional to computability theory.

Consider the equivalence of a von Neumann language, such as the 'while' or 'while'-array language, and a functional language, such as the

induction schemes or cov induction schemes. A von Neumann language defines programs that transform a store or memory; a functional language defines programs that compute functions; thus some care must be taken in defining the meaning of equivalence between programs from the two types of language. In Section 4.3 we defined how functions are computed by von Neumann languages and this was sufficient to define a notion of program equivalence to compare languages: programs are equivalent on $\mathbb{K}$ if they compute the same functions on $\mathbb{K}$.

DEFINITIONS. Let $L = L(\Sigma)$ be a programming language over a class $\mathbb{K}$. We say that L *implements or computes functions over* $\mathbb{K}$ *if there is a subset* L' *of* L *such that to each program* $S \in L'$ *we can uniformly assign a function* f_S^A *on each* $A \in \mathbb{K}$, *and hence assign a function*

$$f_S = \{ f_S^A \mid A \in \mathbb{K} \}$$

on $\mathbb{K}$.

We say that a function f on $\mathbb{K}$ is *implementable or programmable in, or computable by* L, if there exists a program $S \in L$ such that $f = f_S$ on $\mathbb{K}$.

This informal terminology allows us to express the general conventional form of the equivalence theorems seen in this chapter.

FORM FOR EQUIVALENCE THEOREMS: VERSION I. *Let* L_1 *and* L_2 *be programming languages that compute functions over* $\mathbb{K}$. *Let* f *be any function over* $\mathbb{K}$ *Then* f *is computable by* L_1 *iff* f *is computable by* L_2.

To make the equivalence of L_1 and L_2 explicit we need the following definition:

DEFINITION. Let L_1 and L_2 be programming languages that compute functions over $\mathbb{K}$. We say that L_1 and L_2 are *functionally equivalent over* $\mathbb{K}$ if for each $S_1 \in L_1'$ there is $S_2 \in L_2'$ such that $f_{S_1} = f_{S_2}$ on $\mathbb{K}$ and for each $S_2 \in L_2'$ there is $S_1 \in L_1'$ such that $f_{S_2} = f_{S_1}$ on $\mathbb{K}$.

FORM FOR EQUIVALENCE THEOREMS: VERSION II. *Let* L_1 *and* L_2 *be programming languages that compute functions over* $\mathbb{K}$. *Then* L_1 *and* L_2 *are functionally equivalent over* $\mathbb{K}$.

In the theory of programming languages, equivalence theorems should have a richer structure than that traditionally allowed them in computability theory. Further reflection on a theorem having the form of Version II leads us also to make explicit the fact that in its proof two mappings

$$c_1 : L_1' \to L_2' \quad \text{and} \quad c_2 : L_2' \to L_1'$$

are constructed that perform the translations. These mappings are

compilers; and equivalence theorems are theorems about the existence of compilers.

Less obviously, equivalence theorems are also theorems about compiler correctness defined by the notions of program equivalence based upon functional specifications. We will reanalyse Version II using the following concepts.

DEFINITIONS. Let L_1 and L_2 be programming languages that compute functions over $\mathbb{K}$. Let F be a class of functions on $\mathbb{K}$. Then we define programs $S_1 \in L_1'$ and $S_2 \in L_2'$ to be *equivalent on* $\mathbb{K}$ *up to the specifications of* F if there is $f \in F$ such that S_1 and S_2 implement f, *i.e.*

$$f_{S_1} = f_{S_2} = f .$$

In these circumstances, we say S_1 and S_2 are *F-equivalent* and write $S_1 \equiv_F S_2$.

The idea of F-equivalence is used in the concept of compiler correctness as follows:

DEFINITIONS. Let L_1 and L_2 be programming languages that compute functions on $\mathbb{K}$. A map $c: L_1 \to L_2$ is called a *compiler* if it maps syntactically well-formed L_1 programs to syntactically well-formed L_2 programs.

The compiler $c: L_1 \to L_2$ is said to be *correct up to equivalence under the functional specifications of a class of functions F on* $\mathbb{K}$ if for any $S \in L_1'$, if $f_S \in F$ we have $c(S) \equiv_F S$. We will abbreviate this terminology to: c is *correct up to F-equivalence.*

With this terminology we can re-express the general form of equivalence theorems as follows:

FORM FOR EQUIVALENCE THEOREMS: VERSION III (COMPILER VERSION). *Let L_1 and L_2 be languages that compute functions on* $\mathbb{K}$. *Let F be a class of functions on* $\mathbb{K}$. *Then there is a pair of compilers*

$$c_1: L_1 \to L_2 \quad \text{and} \quad c_2: L_2 \to L_1$$

that are correct up to F-equivalence.

The functions can be replaced by relations, and by more general conceptions of program specifications, that lead to more general conceptions of compiler correctness and hence of programming language equivalence.

Version III leads to further enhancements of interest in the theory of programming languages. For example, properties of the compiler, such as its efficiency and the size and efficiency of the programs it returns, can be included.

4.11.3 A second generalization

In the second generalization we will refine Version I of 4.11.1, by clarifying the implementation of functions, and by replacing the hypothesis of the determinism on the language; the account of abstract structures and data types will not be refined. We need the following concepts:

DEFINITIONS. Let L be a programming language that implements functions on $\mathbb{K}$. Let F be a class of functions on $\mathbb{K}$. Then

(*i*) L is *sound* for F if for each program $S \in L'$ the function f_S computed by S is in F;

(*ii*) L is *adequate* for F if for each $f \in F$ there is a program $S \in L'$ that implements f, *i.e.* $f_S = f$;

(*iii*) L is *complete* for F if L is sound and adequate for F.

With this terminology, we can recast the Church-Turing Thesis and its generalizations as statements about soundness. For example, the Church-Turing Thesis can be stated:

> *Any deterministic programming language that implements functions on* $\mathbb{N}$ *is sound for the set of partial recursive functions on* $\mathbb{N}$.

And a commonly used criterion for a general purpose language can be stated as:

> *a programming language is a general purpose language if it is complete for the partial recursive functions on* $\mathbb{N}$.

These statements about $\mathbb{N}$ can be generalized to $\mathbb{K}$ by applying the terminology to Version I; in particular, we have the following revision of Version I:

GENERALIZED CHURCH-TURING THESIS FOR PROGRAMMING LANGUAGES: REVISED VERSION I. *Any deterministic programming language that implements functions on* $\mathbb{K}$ *is sound for the class of cov inductively definable functions on* $\mathbb{K}$.

This thesis is clearly equivalent to Version I. We will now consider some languages of interest and relate them to the thesis. This will suggest a new formulation, Version II, that is worth considering.

Examples of languages that are sound, but are not adequate, are commonplace. For instance, if the **'while'** construct in the **'while'** language is replaced by a bounded iterative construct

do n **times** S **od**

where n is a natural number variable, the new language implements precisely the primitive recursive functions on $\mathbb{N}$.

Examples of languages that are adequate, but are not sound, are necessarily non-deterministic, according to Version I of our Generalized Church-Turing Thesis.

If we replace the assignment statement in the **'while'** language with non-deterministic assignment statements of the form

$$x := y.\Phi(x, y)$$

which assigns a value for y that satisifies the first-order formula Φ; or with the random assignment

$$x := ?$$

then in both cases, under simple and appropriate semantics, the languages implement more than the partial recursive functions on $\mathbb{N}$.

Alternately, if we define the **'while'** language axiomatically, by means of first-order Hoare logic, the semantics obtained is not, in general, the standard semantics of the language and, in particular, need not be either sound or adequate for the partial recursive functions: see Bergstra and Tucker [1984b].

However, it is important to note that some non-deterministic languages are sound for the partial recursive functions and relations, and the cov inductively definable functions and relations. For example, the guarded command language is sound for both under a simple semantics.

It is not appropriate to the concerns of this monograph to attempt an analysis of the ideas of determinism and non-determinism in programming languages that can resolve this conceptual discrepancy with computability. Instead we will formulate a different thesis that we think is better suited to present concerns and that we believe is consistent with Version I; we will replace the hypothesis of determinism with a related hypothesis on the semantics of the language.

GENERALIZED CHURCH-TURING THESIS FOR PROGRAMMING LANGUAGES: VERSION II. *Any programming language that implements functions on* $\mathbb{K}$ *under an operational semantics is sound for the cov inductively definable functions on* $\mathbb{K}$.

Thus we have adopted determinism and operational semantics as the primary concepts delimited by the cov inductively definable functions, in the general setting of abstract data types.

4.11.4 Concluding remarks

Since we wish to program with user-specified abstract data types it is natural to develop a theory of computable functions and relations for abstract data types. The theory presented in this chapter, centered on the cov inductively definable functions on $\mathbb{K}$, is a pleasing and useful generalization of the theory of computable functions on $\mathbb{N}$ that is suitable for abstract data types. In particular, the Generalized Church-Turing Thesis, especially in the form of Version II, can be used to *guide* the technical analysis and evaluation of language constructs in ways similar to those of the Church-Turing Thesis in the restricted case of programming with $\mathbb{N}$ only.

For example, when considering a new language L having abstract data types, we can determine which classes $\mathbb{K}$ are representable in L and equip L with an operational semantics with respect to these classes (if necessary). The Generalized Church-Turing Thesis suggests that L, under its operational semantics, can be compiled into a variety of languages (recall 4.11.2 and Section 4.10). If the language can be shown to be adequate for the cov inductively definable functions then we can expect it to be a general purpose language over $\mathbb{K}$, and to allow a variety of languages to compile into it also.

Further, on the basis of the Thesis, we can expect computations by the language L, under its operational semantics, to be representable by the cov inductively definable functions. This means we can expect to be able to formally specify the input-output behaviour of programs of L in our weak second order assertion language (after the fashion of Section 3.6). We may also expect to be able to devise proof systems for program correctness that are sound and complete.

Starting from these ideas about adequacy and definability of operational semantics in general, we can consider *program equivalence* and questions such as the following: *does the set of all input-output specifications* $\{p\}S\{q\}$ *of a program* S *uniquely determine* S*?* More precisely, let the *partial correctness theory* of S over $\mathbb{K}$ be defined by

$$\mathrm{PC}_{\mathbb{K}}(S) = \{(p,q) \mid \mathbb{K} \models \{p\}S\{q\}\}.$$

Then is it the case that

$$\mathrm{PC}_{\mathbb{K}}(S_1) = \mathrm{PC}_{\mathbb{K}}(S_2) \text{ iff } S_1 \equiv S_2 \text{ over } \mathbb{K}?$$

An extensive study of this question using a first-order assertion language is Bergstra, Tiuryn and Tucker [1982].

These are some of the ways in which the concepts and tools of this monograph can find applications in many areas of programming language theory involving abstract data types.

Bibliography

Ada (1983). *The Programming Language Ada Reference Manual*, American National Standards Institute Inc. ANSI/MIL-STD-1815A-1983, Springer-Verlag, New York.

ADJ (J.A. Goguen, J.W. Thatcher, E.G. Wagner) (1977). An initial approach to the specification, correctness and implementation of abstract data types. 80—149 in: *Current Trends in Programming Methodology, vol. 4: Data Structuring* (ed. R.T. Yeh), Prentice-Hall, Englewood Cliffs, NJ.

P. America, F.S. de Boer (1987). *A proof system for total correctness of recursive procedures.* Centre for Mathematics and Computer Science, Amsterdam, Report. To appear.

K.R. Apt (1981). Ten years of Hoare's logic: A survey — Part I, *ACM Transactions on Programming Languages and Systems*, 3, 431—483.

K.R. Apt (1984). Ten years of Hoare's logic: A survey — Part II: Nondeterminism, *Theoretical Computer Science*, 28, 83—109.

K.R. Apt, G.D. Plotkin (1986). Countable nondeterminism and random assignment, *JACM*, 33, 724—767.

R.J.R. Back (1983). A continuous semantics for unbounded nondeterminism, *Theoretical Computer Science*, 23, 187—210.

J.W. de Bakker (1980). *Mathematical Theory of Program Correctness*, Prentice-Hall International, London.

J. Barwise (1975). *Admissible Sets and Structures*, Springer-Verlag, Berlin, Heidelberg, New York.

J.A. Bergstra, J. Tiuryn, J.V. Tucker (1982). Floyd's principle, correctness theories and program equivalence, *Theoretical Computer Science*, 17, 113—149.

J.A. Bergstra, J.V. Tucker (1982a). Some natural structures which fail to possess a sound and decidable Hoare-like logic for their while-programs, *Theoretical Computer Science*, 17, 303—315.

J.A. Bergstra, J.V. Tucker (1982b). Expressiveness and the completeness of Hoare's logic, *J. Computer and Systems Science*, 25, 267—284.

J.A. Bergstra, J.V. Tucker (1982c). Two theorems about the completeness of Hoare's logic, *Information Processing Letters*, 15, 143—149.

J.A. Bergstra, J.V. Tucker (1982d). The completeness of the algebraic specification methods for data types, *Information and Control*, 54, 186—200.

J.A. Bergstra, J.V. Tucker (1983a). Hoare's logic and Peano's arithmetic, *Theoretical Computer Science*, 22, 265—284.

J.A. Bergstra, J.V. Tucker (1983b). Initial and final algebra semantics for data type specifications: two characterization theorems, *SIAM J. Computing*, 12, 366—387.

J.A. Bergstra, J.V. Tucker (1984a). Hoare's logic for programming languages with two data types, *Theoretical Computer Science*, 28, 215—221.

J.A. Bergstra, J.V. Tucker (1984b). The axiomatic semantics of programs based on Hoare's logic, *Acta Informatica*, 21, 293—320.

S. Brown, D. Gries, T. Szymanski (1972). Program schemes with pushdown stores, *SIAM J. Computing*, 1, 242—268.

A. de Bruin (1984). On the existence of Cook semantics, *SIAM J. Computing*, 13, 1—13.

A. Chandra (1973). *On the properties and applications of program schemas.* Ph.D. Thesis, Department of Computer Science, Stanford University.

A. Chandra, Z. Manna (1975). On the power of programming features, *J. Computer Languages*, 1, 219—232.

A. Church, (1936). An unsolvable problem of elementary number theory, *American J. Math.*, 58, 345—363.

E.M. Clarke, Jr. (1984). The characterization problem for Hoare logic, *Phil. Trans. Royal Soc. London*, A, 312, 423—440.

E.M. Clarke, Jr., S. German, J.Y. Halpern, (1983). Effective axiomatization of Hoare logics, *JACM*, 30, 612—636.

R.L. Constable, D. Gries (1972). On classes of program schemata, *SIAM J. Computing*, 1, 66—118.

S.A. Cook (1976). *Soundness and completeness of an axiom system for program verification.* Department of Computer Science, University of Toronto, Technical Report 95, June 1976.

S.A. Cook (1978). Soundness and completeness of an axiomatic system for program verification, *SIAM J. Computing*, 7, 70—90; Corrigendum 10 (1981), 612.

F. Cristian (1983). Correct and robust programs, *IEEE Transactions on Software Engineering*, **SE**–10, 163—174.

N.J. Cutland (1980). *Computability*, Cambridge University Press, Cambridge.

E.W. Dijkstra (1975). Guarded commands, nondeterminacy and formal derivations of programs, *Communications ACM*, 18, 453—457.

E.W. Dijkstra (1976). *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ.

E. Engeler (1968a). *Formal Languages, Automata and Structures*, Markham Publishing Co., Chicago.

E. Engeler (1968b). Algebraic properties of structures, *Mathematical Systems Theory*, 1, 183—195.

J.E. Fenstad (1975). Computation theories: an axiomatic approach to recursion on general structures. 143—168 in: *Logic Conference, Kiel 1974* (ed. G. Muller, A. Oberschelp, K. Potthoff), Springer-Verlag, Berlin, Heidelberg, New York.

J.E. Fenstad (1980). *Recursion Theory: an Axiomatic Approach*, Springer-Verlag, Berlin, Heidelberg, New York.

R.W. Floyd (1967). Assigning meaning to programs. 19—32 in: *Mathematical Aspects of Computer Science* (ed. J.T. Schwartz), American Mathematical Society, Providence, RI.

H. Friedman (1971). Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory. 361—390 in: *Logic Colloquium '69* (ed. R.O. Gandy, C.E.M. Yates), North-Holland, Amsterdam.

S.J. Garland, D.C. Luckham (1973). Program schemes, recursion schemes and formal languages, *J. Computer and Systems Science*, 7, 119—160.

S. German (1978). Automating proofs of the absence of common runtime errors. 105—118 in: *ACM 5th Symposium on the Principles of Programming Languages, Tucson, Ariz., Jan. 1978*,

J.B. Goodenough (1977). Exception handling: issues and a proposed notation, *Communications ACM*, 18, 683—696.

C.E. Gordon (1970). Comparisons between some generalizations of recursion theory, *Compositio Mathematica* 22, 333—346.

G.A. Gorelick (1975). *A complete axiomatic system for proving assertions about recursive and non-recursive programs*, Department of Computer Science, University of Toronto, Technical Report 75, 1975.

S.A. Greibach (1975). *Theory of Program Structures: Schemes, Semantics, Verification*, Lecture Notes in Computer Science 36, Springer-Verlag, Berlin, Heidelberg, New York.

J.V. Guttag, J.J. Horning (1979). The algebraic specification of data types, *Acta Informatica*, 10, 27—52.

D. Harel (1979). *First-Order Dynamic Logic*, Lecture Notes in Computer Science 68, Springer-Verlag, Berlin, Heidelberg, New York.

C.A.R. Hoare (1969). An axiomatic basis for computer programming, *Communications ACM*, 12, 576—580.

C.A.R. Hoare, N. Wirth (1973). An axiomatic definition of the programming language Pascal, *Acta Informatica*, 2, 335—355.

J. Ichbiah (1983). *Reference Manual for the Ada Programming Language*, Castle House Publications.

C. Jervis (1987). *On the Specification, Implementation and Verification of Data Types*. Ph.D. Thesis, Department of Computer Studies, University of Leeds. In preparation.

S.C. Kleene (1936). *General recursive functions of natural numbers*, *Mathematische Annalen*, 112, 727—742.

S.C. Kleene (1952). *Introduction to Metamathematics*, North-Holland, Amsterdam.

S.C. Kleene (1959). Recursive functionals and quantifiers of finite types I, *Trans. American Math. Society*, 91, 1—52.

S.C. Kleene (1963). Recursive functionals and quantifiers of finite types II, *Trans. American Math. Society*, 108, 106—142.

W.J. Lambert (1968). A notion of effectiveness in arbitrary structures, *J. Symbolic Logic*, 33, 577—602.

H. Ledgard (1981). *Ada: An Introduction and Ada Reference Manuals*, Springer-Verlag, Berlin, Heidelberg, New York.

B.H. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, A. Snyder (1981). *The CLU Reference Manual*, Lecture Notes in Computer Science 144, Springer-Verlag, Berlin, Heidelberg, New York.

B.H. Liskov, A. Snyder (1979). Exception handling in CLU, *IEEE Transactions on Software Engineering*, **SE**–5, 546—558.

B.H. Liskov, S.N. Zilles (1975). Specification techniques for data abstractions, *IEEE Transactions on Software Engineering*, **SE–1**, 7—19.

D. Luckham, D. Park (1964). *The undecidability of the equivalence problem for program schemata.* Bolt, Beranek and Newman Inc., Report 1141.

D. Luckham, D. Park, M.S. Paterson (1970). On formalized computer programs, *J. Computer and Systems Science*, 4, 220—249.

D. Luckham, W. Polak (1980). Ada exception handling: an axiomatic approach, *ACM Transactions on Programming Languages and Systems*, 2, 225—233.

A.R. Martin, J.V. Tucker (1987). The concurrent assignment representation of synchronous systems. 369—386 in: *Parallel Architectures and Languages Europe, Vol. II: Parallel Languages* (ed. J.W. de Bakker, A.J Nijman, P.C. Treleaven), Lecture Notes in Computer Science 259, Springer-Verlag, Berlin, Heidelberg, New York.

Z. Manna (1974). *Mathematical Theory of Computation*, McGraw-Hill.

J. McCarthy (1960). Recursive functions of symbolic expressions and their computation by machine, Part I, *Communications ACM*, 3, 184—195.

J. Meseguer, J.A. Goguen (1985). Initiality, induction and computability. 459—541 in: *Algebraic Methods in Semantics* (ed. M. Nivat and J. Reynolds), Cambridge University Press.

J. Moldestad, J.V. Tucker (1981). *On the classification of computable functions in an abstract setting.* Unpublished manuscript.

J. Moldestad, V. Stoltenberg-Hansen, J.V. Tucker (1980a). Finite algorithmic procedures and inductive definability, *Mathematica Scandinavica*, 46, 62—76.

J. Moldestad, V. Stoltenberg-Hansen, J.V. Tucker (1980b). Finite algorithmic procedures and computation theories, *Mathematica Scandinavica*, 46, 77—94.

R. Montague (1968). Recursion theory as a branch of model theory. 63—86 in: *Logic, Methodology and Philosophy of Science III* (ed. B. van Rootselaar and J.F. Staal), North-Holland, Amsterdam.

Y.N. Moschovakis (1969). Abstract first-order computability I, *Trans. American Math. Society*, 138, 427—464; II: *ibid.*, 138, 465—504.

Y.N. Moschovakis (1971). Axioms for computation theories — first draft. 199—255 in: *Logic Colloquium '69* (ed. R.O. Gandy, C.E.M. Yates), North-Holland, Amsterdam.

D. Normann (1978). Set-recursion. 303—320 in: *Generalized Recursion Theory II: Proc. 1977 Oslo Symposium* (ed. J.E. Fenstad, R.O. Gandy, G.E. Sacks), North-Holland, Amsterdam.

E.-R. Olderog (1983). Expressiveness and the rule of adaptation, *Theoretical Computer Science*, 24, 337—347.

M. Paterson, C. Hewitt (1970). Comparative schematology. 119—128 in: *Record of Project MAC Conference on Concurrent Systems and Parallel Computation*, Association of Computing Machinery, New York.

R.A. Platek (1966). *Foundations of Recursion Theory*. Ph.D. Thesis, Department of Mathematics, Stanford University.

R. Péter (1967). *Recursive Functions*, Academic Press, London and New York.

H. Rogers, Jr. (1967). *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York.

J. Schwarz (1977). Generic commands — a tool for partial correctness formalisms, *Computer J.*, 20, 151—155.

D.S. Scott (1970). Outline of a mathematical theory of computation. 169—176 in: *Proc. 4th Annual Conference on Information Sciences and Systems*, Princeton, NJ.

D.S. Scott, C. Strachey (1971). Towards a mathematical semantics for computer languages. In: *Proc. Symposium on Computers and Automata* (ed. J. Fox), Polytechnic Institute of Brooklyn. Also Technical monograph 6, Programming Research Group, Oxford.

J.C. Shepherdson (1975). Computations over abstract structures: serial and parallel procedures and Friedman's effective definitional schemes. 445—513 in: *Logic Colloquium '73* (ed. H.E. Rose, J.C. Shepherdson), North-Holland, Amsterdam.

J.C. Shepherdson (1985). Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory. 285—308 in: *Harvey Friedman's Research on the Foundations of Mathematics* (ed. L.A. Harrington *et al.*), North-Holland, Amsterdam.

J.C. Shepherdson, H.E. Sturgis (1963). Computability of recursive functions, *JACM*, 10, 217—255.

S. Sokolowski (1977). Total correctness for procedures. 475—483 in: *Proc. 6th Symp. Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 53, Springer-Verlag, Berlin, Heidelberg, New York.

B.C. Thompson (1987) *A Mathematical Theory of Synchronous Concurrent Algorithms.* Ph.D. Thesis, Department of Computer Studies, University of Leeds. In preparation.

B.C. Thompson, J.V. Tucker (1985). Theoretical considerations in algorithm design. 855—878 in: *Fundamental Algorithms for Computer Graphics* (ed. R.A. Earnshaw), Springer-Verlag, Berlin, Heidelberg, New York.

J. Tiuryn (1981). A survey of the logic of effective definitions. 198—245 in: *Logics of Programs: Workshop, ETH Zürich, May—July 1979* (ed. E. Engeler), Lecture Notes in Computer Science 125, Springer-Verlag, Berlin, Heidelberg, New York.

J.V. Tucker (1980). Computing in algebraic systems. 215—235 in: *Recursion theory, its Generalizations and Applications* (ed. F.R. Drake, S.S. Wainer), London Math. Society Lecture Note Series 45, Cambridge University Press.

A.M. Turing (1936-7). On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Society* (2) 42, 230—265. A correction, *ibid.* (2) 43, 544—546. Reprinted in: *The Undecidable* (ed. M. Davis), Raven Press, Hewlett, NY, 1965.

M. Wand (1978). A new incompleteness result for Hoare's system, *JACM*, 25, 168—175.

A. van Wijngaarden (1966). Numerical analysis as an independent science, *BIT*, 6, 68—81.

N. Wirth (1983). *Programming in Modula-2*, Springer-Verlag, Berlin, Heidelberg, New York.

W.A. Wulf (1980), Abstract data types: a retrospective and prospective view. 94—112 in: *Mathematical Foundations of Computer Science: Proc. 9th Symposium, Rydzyna, Poland* (ed. P. Dembinski), Lecture Notes in Computer Science 88, Springer-Verlag, Berlin, Heidelberg, New York.

W.A. Wulf, R.L. London, M. Shaw (1976). An introduction to the construction and verification of Alphard programs, *IEEE Transactions on Software Engineering*, **SE**–2, 253—265.

J.I. Zucker (1980). Expressibility of pre- and postconditions. *Appendix* in: De Bakker (1980), 444-465.