EXPERIMENTS WITH CONTINUATION SEMANTICS: jumps, backtracking, dynamic networks

Arie de Bruin

VRIJE UNIVERSITEIT TE AMSTERDAM

EXPERIMENTS WITH CONTINUATION SEMANTICS:

jumps, backtracking, dynamic networks

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van
doctor in de wiskunde en natuurwetenschappen
aan de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
dr. P.J.D. Drenth,
hoogleraar in de faculteit der sociale wetenschappen,
in het openbaar te verdedigen
op vrijdag 2 mei 1986 te 15.30 uur
in het hoofdgebouw der universiteit,
De Boelelaan 1105

door

ARIE DE BRUIN

geboren te 's-Gravenhage

1986
Centrum voor Wiskunde en Informatica, Amsterdam

Promotor:	prof.dr. J.W. de Bakker
Referent:	dr. K.R. Apt

STELLINGEN

1. Compositionaliteit is een rekbaar begrip.

2. Er is geen scherpe scheidslijn aan te geven tussen operationele en denotationele semantiek die de gangbare opvattingen over deze begrippen recht doet.

3. Het is natuurlijker een specifikatie te geven van het gedrag van een programmamoduul in termen van het input/output gedrag ervan, dan door aan te geven wat het effekt ervan is op waarden van variabelen. Met dergelijke specifikaties als uitgangspunt is het bijvoorbeeld mogelijk een intuitieve redenering te formaliseren tot een korrektheidsbewijs.

   zie: A.P.W. Böhm en A. de Bruin, "Dynamic networks of parallel processes", Report IW 192/82, Mathematisch Centrum, Amsterdam (1982)

4. Het boek van Milne en Strachey overtuigt niet dat de daar gebruikte methode om korrektheid aan te tonen van compilers de aangewezen manier is.

   zie R. Milne en C. Strachey, "A theory of programming language semantics", London: Chapman and Hall, New York, Wiley (2 delen) (1976).

5. Een van de voordelen van de opkomst van de microprocessor was dat het gebruikelijker werd om een proces een eigen processor te verschaffen; het is in dit licht bezien bedenkelijk dat een aantal modernere microprocessoren voorzieningen bieden om multiprogramming efficient te implementeren.

6. Semaforen zijn de **goto** statements van het concurrent programmeren.

7. TEX, het programma van Knuth voor het zetten van teksten, zou nog plezieriger te hanteren zijn als het de mogelijkheid bood een benadering van het eindresultaat zichtbaar te maken op een beeldscherm, en vervolgens verbeteringen zou accepteren die gespecificeerd zijn in termen van dat tussenresultaat.

8. Informatica-opleidingen dienen een praktikum tienvingerig typen in het curriculum op te nemen, al was het maar om te proberen het gebruik van onbegrijpelijke afkortingen aan banden te leggen.

9. Bij de voorlichting over universitaire opleidingen in de informatica dient nadrukkelijk aangegeven te worden dat de studie geen uitgebreide kursus programmeren is, en ook geen opleiding in de bedrijfseconomische of maatschappelijke achtergronden van het vak.

10. Onder invloed van computerspelen zal een nieuwe vorm van kunst ontstaan.

11. Goed leren pianospelen heeft meer te maken met het afleren van overtollige handelingen dan met het aanleren van nieuwe.

12. Het opvoeden van kinderen wordt een stuk interessanter en effektiever als je ook bereid bent je door hen te laten opvoeden.

13. Onverwerkte emoties doen zich graag voor als levenswijsheid.

EXPERIMENTS WITH CONTINUATION SEMANTICS:

jumps, backtracking, dynamic networks

VRIJE UNIVERSITEIT TE AMSTERDAM

EXPERIMENTS WITH CONTINUATION SEMANTICS:

jumps, backtracking, dynamic networks

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van
doctor in de wiskunde en natuurwetenschappen
aan de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
dr. P.J.D. Drenth,
hoogleraar in de faculteit der sociale wetenschappen,
in het openbaar te verdedigen
op vrijdag 2 mei 1986 te 15.30 uur
in het hoofdgebouw der universiteit,
De Boelelaan 1105

door

ARIE DE BRUIN

geboren te 's-Gravenhage

1986
Centrum voor Wiskunde en Informatica, Amsterdam

Promotor:	prof.dr. J.W. de Bakker
Referent:	dr. K.R. Apt

VOOR MIJN OUDERS

ACKNOWLEDGEMENTS

I thank Jaco de Bakker, who was my teacher when I was a student at the VU, my boss when I started working as a researcher at the MC, and who is my promotor now. I have learned much from him, from proving theorems to getting a thesis finished. During this last period his insistence, his insight and his ability to keep things organised have been important for me, and an example.

I thank Krzysztof Apt, the referent of this thesis and my former colleague from the MC. In both roles he has given me the encouragement I needed at some crucial moments, by showing his enthusiasm about my work and his own.

I thank Wim Böhm for being a good guy to work with. During our cooperation I found out that working together can be fun, fruitful and human at the same time.

I thank some of my (former) room mates: Paul Vitányi, who has taught me more of the trade than he possibly realises, Ruurd Kuiper whose enthusiasm (not only about semantics) has been catching, and Cor Bioch, for his insistence on getting things right.

I thank the persons and organisations that founded and maintain the "Stichting Mathematisch Centrum", nowadays called CWI. It is an institute which provides an environment and facilities very well suited for conducting research. I thank my ex-colleagues from the MC, for creating a true scientific atmosphere there,—I have learned a lot from them.

I thank Koos Verhoeff, the head of our computer science department at Erasmus University, for tickling my thoughts on several topics in computer science, and for establishing an atmosphere in the workplace in which I could develop quite a few of my abilities.

I thank my colleagues at Erasmus for their trust, patience and sympathy. I will try to return this when it is their turn.

I thank two programmers: Donald Knuth for making TEX, and Michael Abrash for making Big Top, the ultimate Donkey Kong game.

I thank the people who have cooperated in technical matters: Dick Zwarst and the people from the printing-office at the CWI, Tobias Baanders who designed the cover, Joop van Dijk for making the title page and the figures from chapters 1 and 5, Bart de Vries for his TEXnical assistence, all the typists who have ploughed through all these μ's and $\perp$'s, in particular Cia Scholte during the last stages.

And finally I thank my wife Jos and my daughters Lottie, Margriet and Eva, for being there, sharing my life, insisting on sharing my life, and making it exciting and full. I love you.

CONTENTS

chapter 1

INTRODUCTION

1. INTRODUCTION

1.1. Overview

This thesis contains four papers. In all of them I introduce a (small) programming language and amongst other things present one or more semantics for this language. These papers are reproduced in chapters 2 through 5.

This first chapter consists of three sections. The first one gives an overview of the thesis. The second section presents an introduction to the notions occurring in the rest of this thesis, intended for those who are not very familiar with formal semantics of programming languages. Finally, section 1.3 will provide a more thorough introduction to the material covered in the later chapters, present some conclusions and discuss related work.

The papers in the next four chapters are the following:

A. de Bruin, "On the existence of Cook semantics", *SIAM J. Comput.* **13** (1984), 1–13.

This is chapter 2. The following chapter is a reprint of

A. de Bruin, "Operational and denotational semantics describing the matching process in SNOBOL4", Report IW 151/80, Mathematical Centre, Amsterdam (1980).

Chapter 4 is the paper

A. de Bruin, "**Goto** statements: semantics and deduction systems", *Acta Informatica* **15** (1981), 385–424,

and the final chapter contains

A. de Bruin and A.P.W. Böhm, "The denotational semantics of dynamic networks of processes", *ACM Trans. Prog. Lang. and Syst.* **7** (1985), 656–679.

In the first paper a style of semantics is investigated which has been introduced by

Cook in [Coo78]. His idea is to define the meaning of a program to be a function that takes an initial state as an argument (a state provides the values of all variables), and yields a trace which is a sequence of states. Such a trace consists of all intermediate states generated by execution of the program. If the program does not terminate then the corresponding trace will be an infinite sequence. Cook defined this meaning function through a set of recursive equations. In my paper I investigate whether equations like these always have a well defined solution. To this end I introduce a small language, and show that there exist several ways to establish the intended solution of the corresponding Cook equations.

In the second paper I define three semantics for the SNOBOL4 pattern matcher, and prove them equivalent. The most notable feature to be captured in these semantics is the backtracking mechanism. A pattern match in SNOBOL4 may succeed or fail. In the latter case an alternative will be tried that has been encountered earlier during the match and stacked for later use. This backtracking action is similar to executing a kind of **goto** statement, but there is a difference: this jump has a target that has been determined dynamically in earlier stages of the pattern match. Every step in a pattern match may fail, so in every stage there are two possibilities for proceeding with the computation: a normal one in case the previous step succeeded, and a failure continuation.

The third paper investigates the familiar **goto** statement. In [ClH72] Clint and Hoare have proposed a formal proof system for this statement that is not easily understood. In my paper I justify their system by proving it to be sound and complete. To achieve this result, I need a semantics for the **goto** statement, and in fact I introduce three semantics, one in the style of Cook, a direct denotational one, and a denotational semantics using continuations. After proving the equivalence of these semantics I introduce a more straightforward proof system than the one by Clint and Hoare, which I then justify in a natural way using direct semantics. For the justification of Clint's and Hoare's system it is more appropriate to use continuation semantics.

The final paper deals with the language DNP (dynamic networks of processes) for parallel programming which is a variant of a language introduced in [Kah74, KaM77] by Kahn. In this language one can define processes that are able to expand into networks of several new processes—much in the style of the fork construct in unix. Processes can communicate over channels and these are comparable to unix pipes. Kahn presented an outline of a denotational semantics for the language, but did not formalize this. That is what we do in our paper by presenting a fully worked out denotational semantics.

The latter three papers deal with programming constructs and issues (backtracking, **goto** statements and expansion of a process into a network) that describe a flow of control that is more involved than constructs like the **if-then-else** , the **while** loop or procedure calls discussed by the advocates of structured programming. In section 3 of this introductory chapter I provide some evidence that these concepts are not easily captured by direct denotational semantics, and that continuation semantics is more suitable here. In the first paper on Cook semantics continuations appear on the scene as well: one approach to solve the problem investigated there is to use continuations in constructing a solution to the Cook equations.

So the use of continuations is a common feature in all papers presented here. I found that in some cases continuations were just the tool I needed, they fitted elegantly. In other cases the notion was less adequate, it served its purpose, the problem at hand could be solved by using continuations, but on the other hand introducing them entailed quite some additional complications. In the last section of this chapter I will dwell a little longer on these experiences.

1.2. Formal semantics: an introduction

In the preceding section I mentioned direct denotational semantics and continuations, so the first technical terms have already appeared. This section is meant to provide an introduction to these and related notions. I will not pay much attention to the mathematical backgrounds, my aim here is to show some of the ideas and techniques that are applied in the chapters to follow.

For this purpose I will introduce a very simple programming language, and provide four semantics for it. Each semantics occurring in chapters 2 through 5 is in the style of one of the semantics introduced here and therefore these semantics will serve as a vehicle using which I can explain some of the results from the later chapters.

1.2.1. THE LANGUAGE WHILE

The language has this name, because its most complicated construct is the **while** loop. The syntax of WHILE, in a BNF-like notation, is given by

$$\mathtt{S} \;::=\; \mathtt{x}:=\mathtt{t} \mid \mathtt{S}_1;\mathtt{S}_2 \mid \textbf{if } \mathtt{b} \textbf{ then } \mathtt{S}_1 \textbf{ else } \mathtt{S}_2 \mid \textbf{while } \mathtt{b} \textbf{ do } \mathtt{S}$$

Nothing spectacular: the language has a simple assignment statement, composition of statements, conditional statements and the **while** statement. I give a few remarks on this definition, the style of which is customary in denotational semantics.

As a starting point of this definition there are a few "syntactic classes". First of all, there is the class **Var** which contains the variables. Elements of **Var** are denoted by the letters x and y, possibly with indices. Other classes are **Exp**, containing the (arithmetic) expressions, with elements s and t, and the class **Bexp**, the boolean expressions, with elements b. From these classes the BNF-rule above defines the class **Stat**, the statements with elements S.

For our purposes it does not matter what arithmetic and boolean expressions look like. One can think of the standard pascal expressions, apart from the fact that function calls inside expressions are not allowed. This precludes the possibility of expression evaluation that does not terminate or establishes side effects.

1.2.2. DIRECT DENOTATIONAL SEMANTICS

The first semantics that I will define for WHILE is a direct denotational semantics. The idea is to define a "meaning function" **M**, a valuation that takes a syntactical entity Δ as an argument, for instance an expression or a statement, and yields the denotation of the syntactic element—the meaning Δ denotes. Therefore **M** is a function mapping syntactic domains (e.g. **Exp** or **Stat**) to semantic domains. Denotational semantics is also termed mathematical semantics, and this name alludes to the fact that these semantic domains are mathematically well defined entities, in general function spaces.

As a starting point of this denotational semantics I will define the meaning of arithmetic expressions. So I must define a function with functionality:

$$\mathbf{M}\colon \mathbf{Exp} \rightarrow \text{"meanings"}$$

which maps expressions to their denotations. I write $\mathbf{M}[\![\mathtt{t}]\!]$ for the meaning of $\mathtt{t}$. This introduces a notational convention: when the argument of a function is a syntactic entity it will be enclosed in $[\![.]\!]$-type brackets.

In the end evaluation of an expression will yield a value. It is not necessary here to specify exactly what these values will be, it is sufficient to assume the existence of a set V of possible values, the elements of which will be denoted by symbols α. The meaning of an expression $\mathtt{t}$ cannot however be simply a value from V, because in general the result of evaluation of $\mathtt{t}$ depends on the value of the variables occurring in $\mathtt{t}$.

So I shall use abstraction here, a well known technique in denotational semantics: if $\mathbf{M}[\![\mathtt{t}]\!]$ depends on the value of some variables then make $\mathbf{M}[\![\mathtt{t}]\!]$ a function which takes these values (in some form or another) as an argument and yields the right outcome. To this end I introduce the notion of a state. A state models the values the variables have at a certain moment during execution of a program. The set of all states is denoted by Σ, and a typical element of Σ is written as σ. States can be defined elegantly as functions from **Var** to V, that is $\sigma[\![\mathrm{x}]\!]$ denotes the value of x in state σ.

Combining all this, the result is that **M** is a function

$$\mathbf{M}\colon \mathbf{Exp} \rightarrow \Sigma \rightarrow V,$$

and that the value of $\mathtt{t}$ in σ is denoted by $\mathbf{M}[\![\mathtt{t}]\!]\sigma$. It can be proved that the functionality of **M** given above is equivalent to $\mathbf{M}\colon \mathbf{Exp} \times V \rightarrow \Sigma$, which is of a more usual type. However, the variant chosen here will lead to more elegant clauses in the semantical definitions to follow.

The above formula again introduces some new notation. In denotational semantics higher order functions are used often. These are functions that take functions as an argument or yield a function as a result. Thus it is possible that a function f, when applied to an argument a, yields another function that can be applied to an argument b to yield yet another function. Standard notation would lead to formulae like $((f(a))(b))(c)$, which are hardly pleasing to the eye. There is a convention for this which states that function application should be performed from left to right. Using this convention the above formula can be rewritten as $fabc$ or, another example, $\mathbf{M}[\![\mathtt{t}]\!]\sigma$ should be read as $(\mathbf{M}[\![\mathtt{t}]\!])(\sigma)$. All this corresponds to another convention on the use of the $\rightarrow$ in constructing function domains like $A \rightarrow B$, the set of all functions from A to B. This operator should be evaluated from right to left, i.e. $\mathbf{Exp} \rightarrow \Sigma \rightarrow V$ should be read as $\mathbf{Exp} \rightarrow (\Sigma \rightarrow V)$.

As I did not bother to specify the syntax, I also refrain from defining the meaning function $\mathbf{M}$ of expressions.

Boolean expressions are treated like arithmetic ones, apart from the fact that these expressions yield boolean values. To this end I define the set $W = \{tt, ff\}$, and now the valuation

$$\mathbf{M}\colon \mathbf{Bexp} \rightarrow \Sigma \rightarrow W$$

can be introduced, giving the meaning of boolean expressions. Although I use the same name $\mathbf{M}$ for this function, this will not cause ambiguities because the form of the argument of $\mathbf{M}$ will always be a clear indication as to which variant of $\mathbf{M}$ is intended.

I now investigate the meaning of statements. What is the effect of executing a statement? A popular idea is to use the fact that a program consumes input values and produces output values. This would lead to a meaning function $\mathbf{M}$ such that $\mathbf{M}[\![\mathtt{S}]\!]$ is a function from sequences of input values to sequences of output values. In fact, that is the approach we will take in chapter 5 in defining the semantics of DNP.

However, a similar approach cannot work for WHILE because this language has no I/O-statements. What does work is a related idea: consider as "input" for a statement $\mathtt{S}$ the values of the variables before execution and consider "output" to be the values of the variables after execution. The effect of executing a statement $\mathtt{S}$ is then that the initial values of the variables are transformed into final values. If these initial values are modelled by a state σ, and the final values by a state σ', then the meaning $\mathbf{M}[\![\mathtt{S}]\!]$ of $\mathtt{S}$ is the state transformation that maps σ into σ'. So $\mathbf{M}$ must have functionality $\mathbf{M}\colon \mathbf{Stat} \rightarrow \Sigma \rightarrow \Sigma$.

There is a complication here, because it is possible to specify in WHILE a nonterminating computation. There is no final state for such a computation. Therefore a special value $\perp$ ("bottom") is added to Σ, which yields the extended set of states $\Sigma_\perp$ defined by $\Sigma_\perp = (\mathbf{Var} \rightarrow V) \cup \{\perp\}$. This new value denotes the "result of a nonterminating computation": if $\mathtt{S}$ is a statement that does not terminate if evaluation starts in σ, then we have $\mathbf{M}[\![\mathtt{S}]\!]\sigma = \perp$. A consequence of this is that now the functionality of $\mathbf{M}$ is different, we get

$$\mathbf{M}\colon \mathbf{Stat} \rightarrow \Sigma_\perp \rightarrow \Sigma_\perp .$$

It is however convenient to maintain the convention that σ ranges over the set of proper states, that is over Σ (i.e. σ cannot be equal to $\perp$), unless explicitly stated otherwise.

Now it is possible to define the meaning of the assignment statement "$\mathtt{x := t}$". Suppose this statement is executed in an initial state denoted by σ. Then the final state $\sigma' = \mathbf{M}[\![\, \mathtt{x := t} \,]\!]\sigma$ shall be σ, with the only difference that the value of x has been changed into a new value α (remember, side effects are not allowed in WHILE). This value α has been obtained by evaluating $\mathtt{t}$ in state σ, so $\alpha = \mathbf{M}[\![\mathtt{t}]\!]\sigma$. In other words, we have that $\sigma'[\![\mathtt{y}]\!] = \sigma[\![\mathtt{y}]\!]$ if $\mathtt{y} \not\equiv \mathtt{x}$ and $\sigma'[\![\mathtt{x}]\!] = \mathbf{M}[\![\mathtt{t}]\!]\sigma$. (Here the symbol $\equiv$ stands for "syntactic identity", i.e. if for two syntactic objects Δ and Δ' we have $\Delta \equiv \Delta'$, then Δ and Δ' must be the same sequence of symbols.) There is a notation for states like the one just defined: we write $\sigma' = \sigma[\mathbf{M}[\![\mathtt{t}]\!]\sigma/x]$. More generally, if $f: X \rightarrow A$ is a function and $x \in X$ and $a \in A$, then the "variant" $f[a/x]$ of f is defined through

$$f[a/x] = \begin{cases} f(y), & \text{for all } y \neq x \\ a, & \text{otherwise} \end{cases}$$

The meaning of a conditional statement is also straightforward. Consider evaluation of the statement $\mathtt{if\ b\ then\ S_1\ else\ S_2}$, starting in state σ. The final state after evaluation of the conditional statement depends on the value of the boolean expression $\mathtt{b}$ in σ. If this yields true, then $\mathtt{S_1}$ will be executed and the result will be $\mathbf{M}[\![\mathtt{S_1}]\!]\sigma$, otherwise the result will be $\mathbf{M}[\![\mathtt{S_2}]\!]\sigma$.

All this can be summarized in

$$\mathbf{M}[\![\, \mathtt{if\ b\ then\ S_1\ else\ S_2} \,]\!]\sigma = \begin{cases} \mathbf{M}[\![\mathtt{S_1}]\!]\sigma, & \text{if } \mathbf{M}[\![\mathtt{b}]\!]\sigma = tt \\ \mathbf{M}[\![\mathtt{S_2}]\!]\sigma, & \text{otherwise} \end{cases}$$

A shorter notation is possible:

$$\mathbf{M}[\![\, \mathtt{if\ b\ then\ S_1\ else\ S_2} \,]\!]\sigma = (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \ \rightarrow\ \mathbf{M}[\![\mathtt{S_1}]\!]\sigma,\ \mathbf{M}[\![\mathtt{S_2}]\!]\sigma$$

Later I shall also employ an extended form of this notation: $\alpha_1 \rightarrow \beta_1,\ \alpha_2 \rightarrow \beta_2,\ \ldots\ \alpha_n \rightarrow \beta_n,\ \gamma$, which means "if α_1 then β_1 else if α_2 then β_2 else if $\ldots$ else if α_n then β_n else γ".

Next we treat the meaning of composition of statements. Consider evaluation of $\mathtt{S_1; S_2}$ in initial state σ. This amounts to first evaluating $\mathtt{S_1}$ in σ. Suppose this yields $\sigma' = \mathbf{M}[\![\mathtt{S_1}]\!]\sigma$. Then $\mathtt{S_2}$ must be executed starting in state σ', and this yields $\sigma'' = \mathbf{M}[\![\mathtt{S_2}]\!]\sigma'$. We thus have

$$\sigma'' = \mathbf{M}[\![\, \mathtt{S_1; S_2} \,]\!]\sigma = \mathbf{M}[\![\mathtt{S_2}]\!]\sigma' = \mathbf{M}[\![\mathtt{S_2}]\!](\mathbf{M}[\![\mathtt{S_1}]\!]\sigma),$$

or in other words

$$\mathbf{M}[\![\, \mathtt{S_1; S_2} \,]\!] = \mathbf{M}[\![\mathtt{S_2}]\!] \circ \mathbf{M}[\![\mathtt{S_1}]\!],$$

that is, the denotation of the composition of two statements is the composition of the denotations of these statements.

One must be careful here, because it is possible that evaluation of $\mathtt{S}_1$ does not terminate starting from state σ, that is $\mathbf{M}[\![\mathtt{S}_1]\!]\sigma = \perp$. In that case $\mathbf{M}[\![\,\mathtt{S}_1;\mathtt{S}_2\,]\!]\sigma$ must also be $\perp$, which will be the case if $\mathbf{M}[\![\mathtt{S}_2]\!]\perp = \perp$, that is if $\mathbf{M}[\![\mathtt{S}_2]\!]$ is a so called "strict" function.

Therefore I must be careful enough to arrange the definition of $\mathbf{M}$ such that for all statements $\mathtt{S}$ the state transformation $\mathbf{M}[\![\mathtt{S}]\!]$ is strict. This will be a sufficient condition to capture nonterminating computations. I investigate the clauses derived so far.

In any case, if $\mathbf{M}[\![\mathtt{S}_1]\!]$ and $\mathbf{M}[\![\mathtt{S}_2]\!]$ are strict functions, then so is $\mathbf{M}[\![\,\mathtt{S}_1;\mathtt{S}_2\,]\!]$, which can easily be verified. Investigation of the clause on the assignment statement leads to a more careful definition of the variant of a state: if $\mathbf{M}[\![\mathtt{x} := \mathbf{t}]\!]$ must be strict then we must have that $\perp[\alpha/x] = \perp$ for all α and x. A closer look at the clause on the conditional statement reveals another flaw, we must stipulate $\mathbf{M}[\![\,\mathtt{if\ b\ then\ S_1\ else\ S_2}\,]\!]\perp = \perp$ as well.

From these three observations it can be derived that $\mathbf{M}[\![\mathtt{S}]\!]$ is strict for all statements $\mathtt{S}$ built up from assignment statements using conditional statements and composition of statements only. The proof is by "structural induction", that is the induction step amounts to showing that the theorem holds for a composite statement provided that it is true for its constituent parts.

The last clause in the definition of $\mathbf{M}[\![\mathtt{S}]\!]$, the case where $\mathtt{S}$ is a while statement is the hard one. It is through the while statement that nontermination is introduced in our language. The problem is that there is no straightforward inductive definition of $\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$. The definition to be given below is based on "unwinding the loop": evaluation of `while b do S` is equivalent to evaluating `if b then S; while b do S else skip`, where `skip` stands for the identity statement (we do not bother to formally introduce this construct in our syntax). Whatever $\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$ might be, the following equation must hold:

$$\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!] = \mathbf{M}[\![\,\mathtt{if\ b\ then\ S;\ while\ b\ do\ S\ else\ skip}\,]\!].$$

Using the clauses of $\mathbf{M}$ for composition and conditional statements, and taking $\mathbf{M}[\![\,\mathtt{skip}\,]\!]\sigma = \sigma$, I obtain

$$\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!]\sigma \;=\; (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \rightarrow \mathbf{M}[\![\,\mathtt{S;\ while\ b\ do\ S}\,]\!]\sigma,\ \sigma$$

or

$$\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!]\sigma \;=\; (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \rightarrow \mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!](\mathbf{M}[\![\mathtt{S}]\!]\sigma),\ \sigma.$$

It would be nice if this equation can be used as a definition of $\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$, but that does not work: the equation is circular. It is recursive but not inductive, not all syntactical constructs on the right hand side are simpler than the one on the left hand side.

There is however a standard way to derive the intended state transformation from equations like the one above. The idea is to use this equation to generate a chain of better and better approximations of the function to be derived. As a first approximation ϕ_0 you can take the meaning of the statement **diverge**, the statement that never terminates: $\mathbf{M}[\![\,\texttt{diverge}\,]\!]\sigma = \bot$ for all σ. The second approximation is then derived from the first in a standard way, by substituting the first one for $\mathbf{M}[\![\,\texttt{while b do S}\,]\!]$ in the equation above. This yields

$$\begin{aligned}
\phi_1\sigma &= (\mathbf{M}[\![\texttt{b}]\!]\sigma = tt) \;\rightarrow\; \phi_0(\mathbf{M}[\![\texttt{S}]\!]\sigma),\ \sigma \\
&= (\mathbf{M}[\![\texttt{b}]\!]\sigma = tt) \;\rightarrow\; \mathbf{M}[\![\,\texttt{diverge}\,]\!](\mathbf{M}[\![\texttt{S}]\!]\sigma),\ \sigma \\
&= (\mathbf{M}[\![\texttt{b}]\!]\sigma = tt) \;\rightarrow\; \bot,\ \sigma
\end{aligned}$$

In general, approximation ϕ_{i+1} is obtained from ϕ_i through

$$\phi_{i+1}\sigma \;=\; (\mathbf{M}[\![\texttt{b}]\!]\sigma = tt) \;\rightarrow\; \phi_i(\mathbf{M}[\![\texttt{S}]\!]\sigma),\ \sigma.$$

The functions ϕ_i are equivalent to the meaning of the loop "unwound i times":

```
φ0 = M⟦ diverge ⟧
φ1 = M⟦ if b then S; diverge else skip ⟧
φ2 = M⟦ if b then
             S; (if b then S; diverge else skip)
           else skip ⟧
φ3 = M⟦ if b then
             S; (if b then
                    S; (if b then S; diverge else skip)
                  else skip)
           else skip ⟧
```

etc.

This means that where $\phi_i\sigma$ is defined (i.e. $\phi_i\sigma \neq \bot$), that $\phi_i\sigma = \phi_j\sigma$ for all $j > i$ and also $\phi_i\sigma$ is the final state resulting from evaluating **while b do S** in σ. So indeed the chain $\phi_0, \phi_1, \ldots$ gets more and more defined and approximates the meaning we would like to give to **while b do S**.

Now define $\mathbf{M}[\![\,\texttt{while b do S}\,]\!]$ to be the "limit" of this chain of approximations. That is, put $\mathbf{M}[\![\,\texttt{while b do S}\,]\!]\sigma = \sigma'$ if there is a k such that $\phi_k\sigma = \sigma'$, and put $\mathbf{M}[\![\,\texttt{while b do S}\,]\!]\sigma = \bot$ if there is no such k, that is if for all k we have $\phi_k\sigma = \bot$.

I mentioned approximations and limits which suggests that there is a mathematical justification for all this, and in fact there is. Taking Scott's work as a starting point one can construct domains which have enough structure to allow a rigorous formal treatment of the above construction. In this framework it can also be proved that the limit obtained from equations like the circular one above is a solution of that equation. I promised to stay away from too much mathematics, so I will not pursue this further here. One should consult [Sco70] for the theory, or an introduction to this like e.g. [Sto77, Bak80]. I will state the main definitions and lemma's of the theory in chapter 4, section 4 of this thesis.

I showed before for all statements $\mathtt{S}$ of WHILE not containing while statements that $\mathbf{M}[\![\mathtt{S}]\!]$ is a strict function. This is also the case for $\mathtt{while\ b\ do\ S}$, provided of course that the meaning $\mathbf{M}[\![\mathtt{S}]\!]$ of the body of the loop is strict. This is a direct consequence of the fact that all approximations are strict state transformations (to be proved by induction on i).

Now that I have finished the definition of $\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$ the meaning function $\mathbf{M}$ has been established completely. I summarize:

DIRECT DENOTATIONAL SEMANTICS OF WHILE

For all $\sigma \in \Sigma_\perp$, I define

$$\begin{aligned}
\mathbf{M}[\![\,\mathtt{x:=t}\,]\!]\sigma &= \sigma[\mathbf{M}[\![\mathtt{t}]\!]\sigma/x] \\
\mathbf{M}[\![\,\mathtt{S_1;S_2}\,]\!]\sigma &= \mathbf{M}[\![\mathtt{S_2}]\!](\mathbf{M}[\![\mathtt{S_1}]\!]\sigma) \\
\mathbf{M}[\![\,\mathtt{if\ b\ then\ S_1\ else\ S_2}\,]\!]\sigma &= (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \mathbf{M}[\![\mathtt{S_1}]\!]\sigma,\ \mathbf{M}[\![\mathtt{S_2}]\!]\sigma \\
\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!]\sigma &= (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!](\mathbf{M}[\![\mathtt{S}]\!]\sigma),\ \sigma
\end{aligned}$$

where the last clause is a short hand notation defining the limit of a chain of approximations to be obtained like I have described above, that is, more formally one could write $\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!]\sigma = \lim_{i\to\infty} \phi_i$, where the ϕ_i are defined by $\phi_0\sigma = \perp$, and $\phi_{i+1}\sigma = (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \phi_i(\mathbf{M}[\![\mathtt{S}]\!]\sigma),\ \sigma$.

1.2.3. COMPOSITIONALITY

The semantics defined above has a few nice properties. First of all, this semantics is denotational, and as I remarked before, in order to deserve this name, it must establish a mapping between two well defined domains, a syntactic and a semantic one. The syntactic domain contains objects like expressions and statements, and the semantic domain consists of functions, e.g. state transformations. For every syntactic entity the meaning function $\mathbf{M}$ yields a well defined semantic object.

Second, the definition of $\mathbf{M}$ is compositional. This means that its definition is by induction, viz. on the structure of the syntactic object being defined. The principle of compositionality states that "the meaning of a compound expression is built up from the meanings of its parts" (for a discussion of this principle, see for instance [Jan83]).

In our case this means that the denotation of e.g. a composition $\mathtt{S_1;S_2}$ depends on $\mathtt{S_1}$ and $\mathtt{S_2}$ only through the denotations $\mathbf{M}[\![\mathtt{S_1}]\!]$ and $\mathbf{M}[\![\mathtt{S_2}]\!]$: $\mathbf{M}[\![\,\mathtt{S_1;S_2}\,]\!] = SE_{comp}(\mathbf{M}[\![\mathtt{S_1}]\!], \mathbf{M}[\![\mathtt{S_2}]\!])$ for some operator SE_{comp}, which does not depend on $\mathtt{S_1}$ and $\mathtt{S_2}$. For, denoting $\mathbf{M}[\![\mathtt{S_1}]\!]$ by ϕ_1 and $\mathbf{M}[\![\mathtt{S_2}]\!]$ by ϕ_2, we have that $\mathbf{M}[\![\,\mathtt{S_1;S_2}\,]\!]\sigma =$

$\phi_2(\phi_1\sigma)$. So SE_{comp} is nothing but the composition operator $\circ$, which indeed does not depend on $\mathtt{S}_1$ or $\mathtt{S}_2$.

The same principle applies for the definition of the conditional statement and the while statement, although for the latter case this might not be clear at first sight. However, if you define $\mathbf{M}[\![\mathtt{S}]\!] = \phi$, $\mathbf{M}[\![\mathtt{b}]\!] = \rho$, and the function to be defined: $\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!] = \psi$, then the last clause of our definition of $\mathbf{M}$ becomes

$$\psi\sigma = (\rho\sigma = tt) \rightarrow \psi(\phi\sigma),\ \sigma.$$

This equation does not mention syntactic constructs any more. Furthermore this equation is the only formula which is used in constructing the chain of approximations for $\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$. So, although the definition of $\mathbf{M}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$ is rather involved, it is fully compositional—it uses only the meanings of $\mathtt{b}$ and $\mathtt{S}$.

Let us discuss this from another point of view. The syntactic definition of WHILE can be thought of as describing a set of atomic statements (the assignment statements), and three syntactic constructors which take one or two statements and combine these into a composite statement. I attach names to these constructors, defining

$SY_{comp}(\mathtt{S}_1, \mathtt{S}_2) \equiv \mathtt{S}_1; \mathtt{S}_2$
$SY_{cond}(\mathtt{b}, \mathtt{S}_1, \mathtt{S}_2) \equiv \mathtt{if\ b\ then\ S_1\ else\ S_2}$
$SY_{while}(\mathtt{b},\ \mathtt{S}) \equiv \mathtt{while\ b\ do\ S}$

A semantics for WHILE is compositional if for each syntactic constructor SY_α there exists a corresponding semantic constructor SE_α, such that the meaning of a composite construct $SY_\alpha(\mathtt{S}_1,\ \mathtt{S}_2,\ \ldots)$ can be obtained by applying SE_α to the meanings $\mathbf{M}[\![\mathtt{S}_1]\!], \mathbf{M}[\![\mathtt{S}_2]\!], \ldots$ of its parts. In other words, for every syntactic constructor there must be a commutative diagram

$$\begin{tikzcd}
\mathtt{S}_1, \mathtt{S}_2, \ldots \arrow[r, "\mathbf{M}"] \arrow[d, "SY_\alpha"] & \mathbf{M}[\![\mathtt{S}_1]\!], \mathbf{M}[\![\mathtt{S}_2]\!], \ldots \arrow[d, "SE_\alpha"] \\
SY_\alpha(\mathtt{S}_1, \mathtt{S}_2, \ldots) \arrow[r, "\mathbf{M}"] & \mathbf{M}[\![\, SY_\alpha(\mathtt{S}_1, \mathtt{S}_2, \ldots)\,]\!] = SE_\alpha(\mathbf{M}[\![\mathtt{S}_1]\!], \mathbf{M}[\![\mathtt{S}_2]\!], \ldots)
\end{tikzcd}$$

NB. There is a little flaw here, which is related to the fact that we assumed the assignment statements to be atomic building blocks. However, assignments are composite too: $\mathtt{x:=t}$ is built up from the constituents $\mathtt{x}$ and $\mathtt{t}$. If the definition were fully compositional then there has to be an operator SE_{ass} yielding $\mathbf{M}[\![\,\mathtt{x:=t}\,]\!]$ which should take as arguments $\mathbf{M}[\![\mathtt{t}]\!]$ and also the meaning of $\mathtt{x}$. However this meaning $\mathbf{M}[\![\mathtt{x}]\!]$ denotes the value of $\mathtt{x}$, which is not the semantic object needed here. In fact, in the definition of $\mathbf{M}[\![\,\mathtt{x:=t}\,]\!]$, this very $\mathtt{x}$ appears in the right hand side of the defining equation. I allowed names of variables, which are syntactic entities, in my semantic domains. It is probably better (although more involved) to make a distinction between these two objects.

1.2.4. COOK SEMANTICS

The next semantics for WHILE to be introduced in this section is named after Cook, who introduced this kind of semantics in his paper [Coo78]. The idea is that a statement denotes a function not from initial states to final states, but from states to traces, which are sequences consisting of all the intermediate states a computation goes through. In this way more information about the computation is captured than using direct semantics, and this is relevant in case the computation does not terminate for a given initial state. Direct semantics would yield $\perp$ in such a case, while Cook semantics will yield an infinite sequence of states. Thus Cook semantics is appropriate to capture the behaviour of programs that are not intended to terminate.

A first consequence of this is that using Cook semantics there is no need for a bottom element $\perp$. I therefore use Σ again instead of $\Sigma_\perp$.

Because Cook semantics will manipulate sequences I must introduce some notation. I define Σ^* to be the set of all finite sequences $\langle\sigma_1,\ldots,\sigma_n\rangle$ over Σ with length ≥ 0 (the empty sequence is denoted by $\langle\,\rangle$), Σ^ω denotes the set of all infinite sequences $\langle\sigma_1,\sigma_2,\ldots\rangle$ over Σ, and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. I will use the operator $^\wedge$ for concatenation of sequences and define, for $\tau_1,\tau_2 \in \Sigma^\infty$:

if $\tau_1 = \langle\sigma_1,\ldots\sigma_n\rangle \in \Sigma^*$ and $\tau_2 = \langle\sigma'_1,\ldots\sigma'_m\rangle \in \Sigma^*$,
    then $\tau_1{}^\wedge\tau_2 = \langle\sigma_1,\ldots,\sigma_n,\sigma'_1,\ldots,\sigma'_m\rangle \in \Sigma^*$.
if $\tau_1 = \langle\sigma_1,\ldots\sigma_n\rangle \in \Sigma^*$ and $\tau_2 = \langle\sigma'_1,\sigma'_2,\ldots\rangle \in \Sigma^\omega$,
    then $\tau_1{}^\wedge\tau_2 = \langle\sigma_1,\ldots,\sigma_n,\sigma'_1,\sigma'_2,\ldots\rangle \in \Sigma^\omega$.
if $\tau_1 \in \Sigma^\omega$ and $\tau_2 \in \Sigma^\infty$ then $\tau_1{}^\wedge\tau_2 = \tau_1$

Finally I shall need a function κ that extracts the last element from a sequence, that is $\kappa\langle\sigma_1,\ldots\sigma_n\rangle = \sigma_n$ (NB. the value of $\kappa\tau$ in case τ is an infinite sequence or the empty sequence is irrelevant, cf. chapter 2, section 1, Lemma 1.1).

The semantic function used to define the Cook semantics for WHILE will be denoted by $\mathbf{C}$, and this function has functionality

$$\mathbf{C}\colon \mathbf{Stat} \to \Sigma \to \Sigma^\infty.$$

I will give the clauses of the definition of $\mathbf{C}$ first, and then give a discussion of this definition.

COOK SEMANTICS FOR WHILE

$\mathbf{C}[\![\,\mathtt{x}:=\mathtt{t}\,]\!]\sigma = \langle\sigma[\mathbf{M}[\![\mathtt{t}]\!]\sigma/\mathtt{x}]\rangle$
$\mathbf{C}[\![\,\mathtt{S}_1;\mathtt{S}_2\,]\!]\sigma = \mathbf{C}[\![\mathtt{S}_1]\!]\sigma\ ^\wedge\ \mathbf{C}[\![\mathtt{S}_2]\!](\kappa(\mathbf{C}[\![\mathtt{S}_1]\!]\sigma))$
$\mathbf{C}[\![\,\mathtt{if\ b\ then\ S}_1\ \mathtt{else\ S}_2\,]\!]\sigma = (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\to\; \langle\sigma\rangle^\wedge\mathbf{C}[\![\mathtt{S}_1]\!]\sigma,\;\; \langle\sigma\rangle^\wedge\mathbf{C}[\![\mathtt{S}_2]\!]\sigma$
$\mathbf{C}[\![\,\mathtt{while\ b\ do\ S}\,]\!]\sigma =$

$$(\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \langle\sigma\rangle{}^\wedge\mathbf{C}[\![\mathtt{S}]\!]\sigma{}^\wedge\mathbf{C}[\![\,\mathtt{while\ b\ do\ S}\,]\!](\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma)),\ \langle\sigma\rangle$$

First the assignment. This is straightforward, there are no intermediate stages, so assignment yields a single element sequence consisting of the final state only.

Second, composition. This yields a sequence of states consisting of two parts. The first part corresponds to evaluation of $\mathtt{S}_1$ starting in state σ, and the second part corresponds to evaluation of $\mathtt{S}_2$ from initial state σ' which is the last element of $\mathbf{C}[\![\mathtt{S}_1]\!]\sigma$, the final state resulting from evaluation of $\mathtt{S}_1$. Notice that this second sequence will disappear if $\mathbf{C}[\![\mathtt{S}_1]\!]\sigma$ is infinite, due to the definition of concatenation.

Third, conditional statements. The resulting sequence of states consists of first the initial state, which denotes the stage in which the boolean expression is evaluated, followed by a sequence corresponding to evaluation of $\mathtt{S}_1$ or $\mathtt{S}_2$, depending on the value of the guard $\mathtt{b}$ in σ.

Finally, the while statement. This is analogous to part three, apart from the fact that this definition is circular. In that respect this clause is like the definition of the direct semantics of the while statement.

A Cook definition has a rather operational flavour. One is tempted to use these equations as a device for constructing traces. For instance, take the program

$\mathtt{S} \equiv \mathtt{x}{:=}1;\, \mathtt{S}'$

with

$\mathtt{S}' \equiv \mathtt{while\ x}>0\ \mathtt{do\ x}{:=}\mathtt{x}-1.$

The Cook equations can be used to construct the trace $\mathbf{C}[\![\mathtt{S}]\!]\sigma$ as follows

$$\begin{array}{ll}
\mathbf{C}[\![\,\mathtt{x}{:=}1;\,\mathtt{S}'\,]\!]\sigma = & \text{(clause 2)}\\
\quad = \mathbf{C}[\![\,\mathtt{x}{:=}1\,]\!]\sigma \;^\wedge\; \mathbf{C}[\![\mathtt{S}']\!](\kappa(\mathbf{C}[\![\,\mathtt{x}{:=}1\,]\!]\sigma)) = & \text{(clause 1)}\\
\quad = \langle\sigma[1/\mathtt{x}]\rangle \;^\wedge\; \mathbf{C}[\![\,\mathtt{while\ x}>0\ \mathtt{do\ x}{:=}\mathtt{x}-1\,]\!](\sigma[1/\mathtt{x}]) = & \text{(clause 4)}\\
\quad = \langle\sigma[1/\mathtt{x}]\rangle \;^\wedge & \\
\qquad \Big((\mathbf{M}[\![\mathtt{x}>0]\!](\sigma[1/\mathtt{x}]) \;=\; tt) \;\rightarrow & \\
\qquad\quad \langle\sigma[1/\mathtt{x}]\rangle \;^\wedge\; \mathbf{C}[\![\,\mathtt{x}{:=}\mathtt{x}-1\,]\!](\sigma[1/\mathtt{x}]) \;^\wedge & \\
\qquad\qquad ^\wedge\; \mathbf{C}[\![\mathtt{S}']\!](\kappa(\mathbf{C}[\![\,\mathtt{x}{:=}\mathtt{x}-1\,]\!](\sigma[1/\mathtt{x}]))), & \\
\qquad\quad \langle\sigma[1/\mathtt{x}]\rangle\,\Big) & \text{(def of } \mathbf{M})\\
\quad = \langle\sigma[1/\mathtt{x}],\ \sigma[1/\mathtt{x}]\rangle \;^\wedge\; \mathbf{C}[\![\,\mathtt{x}{:=}\mathtt{x}-1\,]\!](\sigma[1/\mathtt{x}]) \;^\wedge & \\
\qquad\qquad ^\wedge\; \mathbf{C}[\![\mathtt{S}']\!](\kappa(\mathbf{C}[\![\,\mathtt{x}{:=}\mathtt{x}-1\,]\!](\sigma[1/\mathtt{x}]))) = & \text{(clause 1)}\\
\quad = \langle\sigma[1/\mathtt{x}],\ \sigma[1/\mathtt{x}],\ (\sigma[1/\mathtt{x}])[0/\mathtt{x}]\rangle \;^\wedge & \\
\qquad\qquad ^\wedge\; \mathbf{C}[\![\mathtt{S}']\!]((\sigma[1/\mathtt{x}])[0/\mathtt{x}]) \;= & \text{(def } \sigma[\alpha/\mathtt{x}])\\
\quad = \langle\sigma[1/\mathtt{x}],\ \sigma[1/\mathtt{x}],\ \sigma[0/\mathtt{x}]\rangle \;^\wedge & \\
\qquad\qquad \mathbf{C}[\![\,\mathtt{while\ x}>0\ \mathtt{do\ x}{:=}\mathtt{x}-1\,]\!](\sigma[0/\mathtt{x}]) = & \text{(clause 4)}\\
\quad = \langle\sigma[1/\mathtt{x}],\ \sigma[1/\mathtt{x}],\ \sigma[0/\mathtt{x}]\rangle \;^\wedge & \\
\qquad \Big((\mathbf{M}[\![\mathtt{x}>0]\!](\sigma[0/\mathtt{x}]) \;=\; tt) \;\rightarrow\; \langle\sigma[0/\mathtt{x}]\rangle^\wedge\ldots,\ \langle\sigma[0/\mathtt{x}]\rangle\Big) \;= & \text{(def of } \mathbf{M})\\
\quad = \langle\sigma[1/\mathtt{x}],\ \sigma[1/\mathtt{x}],\ \sigma[0/\mathtt{x}],\ \sigma[0/\mathtt{x}]\rangle. &
\end{array}$$

It was Cook's intention that these equations be used in such an operational manner, viz. to construct traces the way I just did. One should read the quotation from [Coo78] in chapter 2, section 1.1 on that matter.

However on closer inspection it turns out that this semantics is fully denotational: first of all $\mathbf{C}[\![\mathtt{S}]\!]$ is a well defined object for every $\mathtt{S} \in \mathbf{Stat}$, and secondly the definition is compositional. These properties can be proven just like I did this for the meaning function $\mathbf{M}$ of direct semantics. Again the hard case is $\mathbf{C}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$ There are two questions: is $\mathbf{C}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$ well defined through the Cook equations, and does the definition of $\mathbf{C}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$ depend on $\mathbf{M}[\![\mathtt{b}]\!]$ and $\mathbf{C}[\![\mathtt{S}]\!]$ only. Again the trick is to unwind the loop in order to generate useful approximations of $\mathbf{C}[\![\,\mathtt{while\ b\ do\ S}\,]\!]\sigma$. In this case it is better not to take the statement **diverge** as a starting point, but the statement **disappear** instead. This statement is defined by $\mathbf{C}[\![\,\mathtt{disappear}\,]\!]\sigma = \langle\,\rangle$, for all σ.

The statement **diverge** would not work for several reasons. First of all we do not have the bottom element at our disposal any more. But more importantly, we want a nonterminating computation to yield an infinite row and this cannot be realised with **diverge** as first approximation (**diverge** models nonterminating computations with $\perp$). Instead we start approximating with a statement that yield no information whatever,—it generates the empty sequence in all cases. As the approximations become better and better a larger initial segment of the "real" outcome will be generated. This can be seen as follows.
The zero-th approximation ϕ_0 is characterized by

$$\phi_0\sigma = \mathbf{C}[\![\,\mathtt{disappear}\,]\!]\sigma = \langle\,\rangle$$

The next approximation can be derived from ϕ_0 in the usual way:

$$\begin{aligned}\phi_1\sigma &= (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \langle\sigma\rangle \mathbin{^\wedge} \mathbf{C}[\![\mathtt{S}]\!]\sigma \mathbin{^\wedge} \phi_0(\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma)),\ \langle\sigma\rangle \\ &= (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \langle\sigma\rangle \mathbin{^\wedge} \mathbf{C}[\![\mathtt{S}]\!]\sigma,\ \langle\sigma\rangle\end{aligned}$$

and in general we have

$$\phi_{i+1}\sigma = (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \langle\sigma\rangle \mathbin{^\wedge} \mathbf{C}[\![\mathtt{S}]\!]\sigma \mathbin{^\wedge} \phi_i(\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma)),\ \langle\sigma\rangle$$

So there is a similar result as in the definition of $\mathbf{M}$:

$$\begin{aligned}\phi_0 &= \mathbf{C}[\![\,\mathtt{disappear}\,]\!] \\ \phi_1 &= \mathbf{C}[\![\,\mathtt{if\ b\ then\ S;\ disappear\ else\ skip}\,]\!] \\ \phi_2 &= \mathbf{C}[\![\,\mathtt{if\ b\ then\ S;\ (if\ b\ then\ S;\ disappear\ else\ skip)} \\ &\qquad\quad \mathtt{else\ skip}\,]\!]\end{aligned}$$

etc.

From these equations it follows that the chain $\phi_0\sigma$, $\phi_1\sigma$, $\phi_2\sigma$, ... will indeed yield longer and longer initial segments of the trace corresponding to evaluation of the statement **while b do S**. This is so, because evaluation according to ϕ_i generates intermediate states just like **while b do S** does until the statement **disappear** is hit upon. Then no more states will be generated. Therefore, the more the loop is unwound, that is the deeper **disappear** is buried inside the **if-then-else**'s, the longer the generated sequence will be.

As before, it can be proved, using Scott domains again, that chains obtained from equations like the one above, always have a limit, and that this limit is a solution of the generator equation, i.e. the last equation of the definition of **C**. Finally, the derivation of ϕ_{i+1} from ϕ_i does depend on **b** and **S** only through $\mathbf{M}[\![\mathtt{b}]\!]$ and $\mathbf{C}[\![\mathtt{S}]\!]$, and therefore the definition of $\mathbf{C}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$ is again compositional.

However, the Cook semantics discussed in this section is an exception in two respects. First of all, most of the Cook-style semantics appearing in the literature are not compositional. The semantics in Cook's original paper is not either, and also the Cook semantics occurring in chapters 2, 3 and 4 of this thesis are not. However, all of these semantics can be redefined so as to be fully compositional.

Secondly, for most languages one cannot prove that **C** is well defined in the same straightforward way as I described here. The fact that the above construction could be justified using Scott domains is a coincidence which has to do with the simplicity of the language WHILE. For languages which are only slightly more complex Scott domains are not powerful enough.

In the next chapter I investigate the Cook semantics of such a language which has parameterless procedures instead of the **while** statement. Justification of the Cook semantics for this language is more complicated, Scott domains do not provide Σ^∞ with enough structure. The most satisfactory solution is given in section 6 of chapter 2. There I use the fact that Σ^∞ can be regarded as a metric topological space. This leads to a notion of convergence that is less restricted than convergence in the sense of Scott, which is sufficient to justify a construction like the one above.

1.2.5. OPERATIONAL SEMANTICS

When I discussed the Cook semantics for WHILE, I remarked that these equations can be used in an operational way and I gave an example of this. I repeatedly used clauses from the Cook equations to determine the value of the formula $\mathbf{C}[\![\,\mathtt{x:=1;\ while\ x>0\ do\ x:=x-1}\,]\!]\sigma$. Thus the Cook equations described a kind of machine. In each step of this machine part of the expression to be evaluated is expanded by "body replacement": a subexpression of the form $\mathbf{C}[\![\mathtt{S}]\!]\sigma$ is replaced by the right hand side of the corresponding Cook equation.

This is the main idea behind operational semantics: define an abstract machine, which can be in some "configuration", and capture the behaviour of such a machine through a function STEP which transforms machine configurations into machine configurations.

A configuration models the interior of a computer at a certain moment during evaluation of a program. Therefore a configuration should contain a state, in order to find out what the values are of the variables, and it must also contain a representation of the part of the program awaiting evaluation. If a computation terminates then, after a number of STEPs, the initial configuration will have been transformed into a final configuration which is characterised by the fact that the program part of this configuration is empty.

This idea has been introduced by Landin in [Lan64], where he defined the SECD machine, intended for evaluation of λ-terms. This semantics is one of the roots of the Vienna Definition Language (cf. e.g. [Weg72]). It also appears in other places [Sto77, ch. 13], [Sto81]. In chapter 3, section 3 of this thesis I will introduce a semantics in this style for a fragment of SNOBOL4. A variant of this semantics can be found in [HeP79], and [Ap81a]. The difference is that here I will define the function STEP by cases, while they introduce a "transition system" which defines, through a system of axioms and rules, the desired relation between configurations.

In order to establish the operational semantics I start by defining the set **Conf** of machine configurations:

$\mathbf{Conf} = (\mathbf{Stat} \cup \{\mathtt{skip}\}) \times \mathbf{Store} \times \mathbf{Stat}^*$,

that is, a configuration $(\mathtt{S}, s, T)$ consists of the statement $\mathtt{S}$ being evaluated, the current store which is modelled by s and a sequence T of statements which will be evaluated once $\mathtt{S}$ has been worked through. The set **Store** corresponds to the set Σ of proper states in the denotational semantics given before, in the sense that both provide values of variables. There is however a fundamental difference: a store s is not a function but a finite object, for instance a list of variable-value pairs, on which two operations are defined: INSERT v x s which yields a new store which is like s apart from the variable x which now has the value v, and RETRIEVE x s, which delivers the value of x in s. Axioms for these operations might be [McC63]:

$\text{RETRIEVE x (INSERT } v \text{ x } s) = v$

$\text{RETRIEVE y (INSERT } v \text{ x } s) = \text{RETRIEVE y } s, \qquad \text{if y} \not\equiv \text{x}.$

A configuration $c = (\mathtt{S}, s, T)$ is called final if $\mathtt{S} \equiv \mathtt{skip}$ and $T = \langle\,\rangle$.

I next define the STEP function assuming that there exists a function VAL such that VAL **t** s (resp. VAL **b** s) yields the value of the (boolean) expression **t** (resp. **b**) in state s:

$\text{STEP}(\mathtt{skip}, s, \langle\,\rangle) = (\mathtt{skip}, s, \langle\,\rangle)$

$\text{STEP}(\mathtt{skip}, s, \langle \mathtt{S}\rangle \widehat{\ } T) = (\mathtt{S}, s, T)$

$\text{STEP}(\mathtt{x} := \mathbf{t}, s, T) = (\mathtt{skip},\ \text{INSERT}\,(\text{VAL } \mathbf{t}\ s)\ \mathtt{x}\ s,\ T)$

$\text{STEP}(\mathtt{S}_1; \mathtt{S}_2, s, T) = (\mathtt{S}_1, s, \langle \mathtt{S}_2\rangle \widehat{\ } T)$

$\text{STEP}(\mathbf{if}\ \mathbf{b}\ \mathbf{then}\ \mathtt{S}_1\ \mathbf{else}\ \mathtt{S}_2, s, T) =$

$\quad (\text{VAL } \mathbf{b}\ s = \text{TRUE}) \ \rightarrow\ (\mathtt{S}_1, s, T),\ (\mathtt{S}_2, s, T)$

$\text{STEP}(\mathbf{while}\ \mathbf{b}\ \mathbf{do}\ \mathtt{S}, s, T) =$

$\quad (\text{VAL } \mathbf{b}\ s = \text{TRUE}) \ \rightarrow\ (\mathtt{S}, s, \langle \mathbf{while}\ \mathbf{b}\ \mathbf{do}\ \mathtt{S}\rangle \ \widehat{\ }\ T),\ (\mathtt{skip}, s, T)$

A few remarks are in order. First of all, STEP does not alter final configurations which is like it should be. This first clause is not really needed though, it is only included for convenience. The statement **skip** has been introduced as a marker indicating that evaluation of the current statement has been finished. Evaluation of an assignment statement leads to an altered state. Evaluation of a composition $\mathtt{S}_1;\mathtt{S}_2$ amounts to storing $\mathtt{S}_2$ for later use on the stack of statements to be evaluated and switching attention to evaluation of $\mathtt{S}_1$. The first step of evaluation of an **if-then-else** statement is determining which branch should be taken. Evaluation of a while statement is similar, but if the guard is true then there are possibly more iterations of the loop to be performed and therefore the while statement must be stacked.

Now I can define the funcion $\mathbf{O}\colon \mathbf{Stat} \to \mathbf{Store} \to \mathbf{Store}$ by the equations

$$\mathbf{O}[\![\mathtt{S}]\!]s = \begin{cases} s', & \text{if there exists configurations } c_0,\ c_1,\ \ldots,\ c_k\ (k \geq 0) \text{ with} \\ & \quad c_0 = (\mathtt{S}, s, \langle\,\rangle) \\ & \quad c_{i+1} = \text{STEP } c_i \quad (0 \leq i \leq k-1) \\ & \quad c_k = (\mathtt{skip}, s', \langle\,\rangle) \text{ is a final configuration} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

There are two important differences between this approach and denotational semantics. First of all this semantics uses finite representations modelling e.g. the program to be evaluated or the values of the variables, while denotational semantics uses infinitary objects, e.g. function spaces on infinite domains. The other difference is that this approach does not yield a denotation "in one step" so to speak. Instead a representation of the original program and the initial state are fed into an abstract machine which will in the end, after a number of steps, produce a final outcome. The function STEP cannot be defined in a compositional way because there are no denotations of parts of a composite syntactical construct. Therefore the denotation of this composite statement is not defined in terms of the denotations of its constituents. The latter remark also applies for $\mathbf{O}$: though $\mathbf{O}[\![\mathtt{S}_1]\!]$ and $\mathbf{O}[\![\mathtt{S}_2]\!]$ are both well defined functions it does not hold that $\mathbf{O}[\![\,\mathtt{S}_1;\mathtt{S}_2\,]\!]$ is defined in terms of $\mathbf{O}[\![\mathtt{S}_1]\!]$ and $\mathbf{O}[\![\mathtt{S}_2]\!]$.

1.2.6. CONTINUATION SEMANTICS

The last semantics to be introduced is continuation semantics. It is again a denotational one, and in fact the central one of this thesis. In direct semantics the denotation for a statement is a function describing how evaluation of the statement transforms an initial state into a final state. From the fact that this semantics is compositional, it follows that such a transformation captures sufficient information about the statement. In order to establish the direct semantics for WHILE no more information about statements was needed, and this is due to the fact that structured statements in WHILE are built up from their parts in a very controlled manner: only statements containing a single entry point and a single exit point are used as building blocks and the result is again a statement with one entry and exit point. For instance the statement **while b do S** is built up from **b** and **S** according to the flow chart of figure 1.

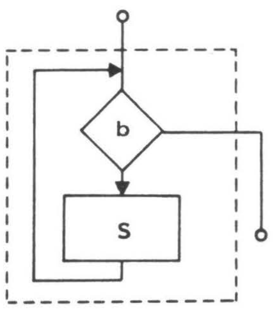

Figure 1. Flowchart of the `while` statement.

Here S might be any statement as long as it has only one entry point and one exit point, and the resulting `while` statement is a "module" with the same properties. The meaning of the `while` statement as suggested by this flow chart is independent of the interior of the module (black box) S. The only information needed about S to establish the meaning of the whole construct is its external behaviour modelled by the corresponding state transformation. This is the "structured programming" approach.

The same idea can be found in Hoare like proof systems. One can express properties of programs in this system using formulae of the form {p}S{q} where p and q are assertions about the initial and final states of the program. Such a property can be proven from similar properties of the constituent statements of S. Thus, Hoare like formulae only describe the external behaviour of a statement.

However, most programming languages do not adhere to this single-entry; single-exit principle. These languages feature constructs like error exits (execution of a program terminates because a run time error has been discovered), **break** statements which terminate execution of a loop, and general **goto** statements. The consequence is that direct semantics is in general not feasible for "full" languages, and another technique has to be applied. This is continuation semantics. Continuations are due to Strachey and Wadsworth [StW74], Morris [Mor70] and Mazurkiewicz[Maz70]. Continuation semantics is often called standard semantics [Sto77, MiS76] because this is the preferred denotational semantics for "real" languages.

I shall treat continuations by introducing error exits into the language WHILE: evaluation of statements can terminate abnormally because a run time error is discovered. More specifically, evaluation of expressions can be broken off for this reason. To this end I extend the value domain V, adding a new value ERROR.

Before giving the solution using continuation semantics I must remark however that it is possible to incorporate such an exit mechanism in direct semantics as well. The method is to extend Σ with a new state ERR, the "error state". The meaning of an assignment statement x:=t in σ should be ERR if $\mathbf{M}[\![\mathtt{t}]\!]\sigma$ yields ERROR. Now, in order to make this technique work, all meanings of statements must be "error strict", that is $\mathbf{M}[\![\mathtt{S}]\!]$ERR must be ERR for all S.

The effect is then that the error is "passed through" from each statement onto the next one until it reaches the last statement of the program which then delivers ERR as a final state. Notice that the same mechanism has been used with $\perp$ describing a nonterminating computation. Care must be taken that these mechanisms do not interfere with each other, e.g. if the body S of the loop `while true do S` generates an error then the loop should terminate yielding ERR. This can be worked out satisfactorily however, and therefore error exits can very well be defined using direct semantics. This technique of coding additional information into the intermediate states can be observed more often. I will use it in chapter 4, at the end of section 4, where I will define a direct semantic function **A** for the **goto** statement. I will discuss this idea again in section 1.3 where also some remarks will be made on the advantages and disadvantages of this direct approach versus the continuation approach. So much for direct semantics here, I now return to continuation semantics.

This can best be introduced by considering a program $\mathtt{S}' \equiv \ldots; \mathtt{S}; \ldots$ containing a statement S. I will call the result of evaluation of $\mathtt{S}'$ an answer and introduce the domain A for this, the elements of which will be denoted by the symbol δ. An answer can be a well defined state in case evaluation of $\mathtt{S}'$ terminates not in error, it can be $\perp$ in case evaluation of $\mathtt{S}'$ gets caught in an infinite loop, or it can be ERR in case an error is discovered. The definition of A is therefore $A = \Sigma \cup \{\mathrm{ERR}, \perp\}$.

Again, like when we worked with Cook semantics, there is no need to pass $\perp$ from statement to statement, so again I will use Σ and not $\Sigma_\perp$.

I will use the symbol **N** for the meaning function of continuation semantics and therefore the result of evaluating the program $\mathtt{S}'$ given above starting in state σ' can be written as $\mathbf{N}[\![\mathtt{S}']\!]\sigma'$ which must be an answer $\delta \in A$. Continuation semantics will again be compositional, and therefore the meaning $\mathbf{N}[\![\mathtt{S}']\!]$ of the program $\mathtt{S}'$ must be built up, among other things, from the meaning $\mathbf{N}[\![\mathtt{S}]\!]$ of S.

Suppose that, during this computation, S will be evaluated starting in some intermediate state σ, i.e. $\mathbf{N}[\![\mathtt{S}]\!]\sigma$ must be determined. The main idea of continuation semantics is that the meaning $\mathbf{N}[\![\mathtt{S}]\!]$ of S applied to this state σ will not be a state σ'' resulting from evaluation of S alone. Instead, $\mathbf{N}[\![\mathtt{S}]\!]\sigma$ will be some answer δ denoting the result of evaluating S followed by the rest of the program $\mathtt{S}'$ that still has to be executed.

Before pondering on how this δ can be obtained, it is worthwile to realise that the problem of an error state being passed from one statement to another has been solved. If S is an assignment $\mathtt{x} := \mathtt{t}$, and evaluation of S in σ results in an error then we just define $\mathbf{N}[\![\mathtt{S}]\!]\sigma = \mathrm{ERR}$, which states that the final answer of evaluating $\mathtt{S}'$ is ERR. Similarly, if evaluation of S does not terminate starting from σ, then $\mathbf{N}[\![\mathtt{S}]\!]\sigma$ yields $\perp$ as the final answer.

The difficult part is of course what the result of $\mathbf{N}[\![\mathtt{S}]\!]$ should be in case evaluation of S in state σ terminates normally in some state σ''. This σ'' cannot be the final answer because σ'' is an intermediate state in the execution of $\mathtt{S}'$ which is in general not finished after $\mathtt{S}$ has been evaluated. As it stands, the final answer δ cannot be obtained because this answer depends on the "future" of the computation, viz. the statements of $\mathtt{S}'$ which have to be evaluated after $\mathtt{S}$ has been executed, and the only information given to the meaning function $\mathbf{N}$ is the statement $\mathtt{S}$ and the initial state σ. To solve this, abstraction must be used again (cf. section 1.2.2): if $\mathbf{N}[\![\mathtt{S}]\!]$ does not only depend on the initial state σ but also on the future of the computation (this future θ will be called a continuation in the future), then make $\mathbf{N}[\![\mathtt{S}]\!]$ a function not only of σ but also of θ. In other words, it makes sense to put $\mathbf{N}[\![\mathtt{S}]\!]\theta\sigma = \delta$, and define the functionality of $\mathbf{N}$ by

$$\mathbf{N}\colon \mathbf{Stat} \rightarrow \text{“continuations”} \rightarrow \Sigma \rightarrow A.$$

In this stage it is worthwile to look again at the operational semantics of the previous section, because similar ideas could be observed there. A configuration $(\mathtt{S}, s, T)$ has to contain enough information to yield the final result of the whole computation which means that there has to be another component apart from the statement $\mathtt{S}$ being evaluated and the initial store s. Information about the future of the computation is needed as well and this is provided by T, the stack containing the statements awaiting execution.

In the semantics of this section it is not feasible to work with such a "syntactic continuation", a list of statements. Instead I will use a denotation θ for this, which is simply the meaning of the corresponding list of statements, that is a continuation θ is a function transforming an initial state (modelling the intermediate state after execution of $\mathtt{S}$) into a final answer.

So we have the following result: if $\mathtt{S}$ is a statement occurring in a program $\mathtt{S}'$ and $\mathtt{S}$ is evaluated in initial state σ, and if the future of the computation is given by the continuation θ, then the final answer of the computation is given by $\mathbf{N}[\![\mathtt{S}]\!]\theta\sigma$. Or, stated a little bit differently, if θ is the function in $\Sigma \rightarrow A$ describing what happens after evaluation of $\mathtt{S}$ then $\mathbf{N}[\![\mathtt{S}]\!]\theta$ is the function in $\Sigma \rightarrow A$ describing what happens if you start at $\mathtt{S}$ instead.

The ideas presented above lead to the following definition:

CONTINUATION SEMANTICS OF WHILE (without error exits)

$$\begin{aligned}
&\mathbf{N}[\![\,\mathtt{x:=t}\,]\!]\theta\sigma = \theta(\sigma[\mathbf{M}[\![\mathtt{t}]\!]\sigma/\mathtt{x}])\\
&\mathbf{N}[\![\,\mathtt{S}_1;\ \mathtt{S}_2\,]\!]\theta\sigma = \mathbf{N}[\![\mathtt{S}_1]\!]\{\mathbf{N}[\![\mathtt{S}_2]\!]\theta\}\sigma\\
&\mathbf{N}[\![\,\mathtt{if\ b\ then\ S_1\ else\ S_2}\,]\!]\theta\sigma \;=\; (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \mathbf{N}[\![\mathtt{S}_1]\!]\theta\sigma,\ \mathbf{N}[\![\mathtt{S}_2]\!]\theta\sigma\\
&\mathbf{N}[\![\,\mathtt{while\ b\ do\ S}\,]\!]\theta\sigma \;=\; (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \mathbf{N}[\![\mathtt{S}]\!]\{\mathbf{N}[\![\,\mathtt{while\ b\ do\ S}\,]\!]\theta\}\sigma,\ \theta\sigma
\end{aligned}$$

A few remarks. Assignment is straightforward, if the rest of the computation is given by θ, then if you start off with an assignment $\mathtt{x:=t}$ first, the contribution of the assignment is that first the initial state σ is transformed into the intermediate state $\sigma[\mathbf{M}[\![\mathtt{t}]\!]\sigma/\mathtt{x}]$, and in order to obtain the final answer the continuation θ must then be applied to this state.

The clause on composition can best be understood by reconsidering my last remark before I gave the definition of $\mathbf{N}$, which can be restated as: if θ is the continuation describing the future of the computation after $\mathtt{S}$ has been evaluated, then $\mathbf{N}[\![\mathtt{S}]\!]\theta$ is again a continuation, now describing the computation if it starts at $\mathtt{S}$. Therefore, if the future after evaluation of $\mathtt{S}_1;\mathtt{S}_2$ is given by θ, then evaluation of $\mathtt{S}_1;\mathtt{S}_2$ followed by θ is equivalent to evaluation of $\mathtt{S}_1$ followed by the future consisting of $\{$evaluation of $\mathtt{S}_2$ followed by $\theta\}$, that is consisting of $\mathbf{N}[\![\mathtt{S}_2]\!]\theta$. I have introduced a notation here: in order to make expressions containing arguments which are continuations more readable, these continuations are often enclosed in $\{.\}$-brackets.

The clause on the conditional statement is straightforward, and the clause on the **while** statement is obtained by unwinding the loop once and then using clauses 2 and 3. This last clause again defines a chain of approximations of $\mathbf{N}[\![\,\mathtt{while\ b\ do\ S}\,]\!]\theta$:

$$\begin{aligned}
&\theta_0\sigma = \mathbf{N}[\![\mathtt{diverge}]\!]\theta\sigma\\
&\theta_1\sigma = (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \mathbf{N}[\![\mathtt{S}]\!]\theta_0\sigma,\ \theta\sigma\\
&\theta_2\sigma = (\mathbf{M}[\![\mathtt{b}]\!]\sigma = tt) \;\rightarrow\; \mathbf{N}[\![\mathtt{S}]\!]\theta_1\sigma,\ \theta\sigma,
\end{aligned}$$

where $\mathbf{N}[\![\mathtt{diverge}]\!]\theta\sigma = \bot$ for all θ and σ. Notice that $\mathbf{N}[\![\,\mathtt{while\ b\ do\ S}\,]\!]\theta$ is approximated here, not $\mathbf{N}[\![\,\mathtt{while\ b\ do\ S}\,]\!]$. More syntactically, this can also be written as:

$$\begin{aligned}
&\theta_0 = \mathbf{N}[\![\,\mathtt{diverge}\,]\!]\theta\\
&\theta_1 = \mathbf{N}[\![\,\mathtt{if\ b\ then\ S;\ diverge\ else\ skip}\,]\!]\theta\\
&\theta_2 = \mathbf{N}[\![\,\mathtt{if\ b\ then\ S;\ (if\ b\ then\ S;\ diverge\ else\ skip)\ else\ skip}\,]\!]\theta
\end{aligned}$$

etc. It is instructive to check how nonterminating loops are handled now. Consider for instance the statement $\mathtt{while\ true\ do\ skip;\ \ x{:=}0}$.

The meaning of this statement with respect to a continuation θ is, according to the second clause of the definition of $\mathbf{N}$:

$$\begin{aligned}
&\mathbf{N}[\![\,\mathtt{while\ true\ do\ skip;\ \ x{:=}0}\,]\!]\theta\sigma\ =\\
&\qquad = \mathbf{N}[\![\,\mathtt{while\ true\ do\ skip}\,]\!]\,\{\mathbf{N}[\![\,\mathtt{x{:=}0}\,]\!]\theta\}\,\sigma.
\end{aligned}$$

Consider the approximation chain of this latter formula. We get:

$$\begin{aligned}
\theta_0\sigma &= \mathbf{N}[\![\,\mathtt{diverge}\,]\!]\,\{\mathbf{N}[\![\,\mathtt{x{:=}0}\,]\!]\theta\}\,\sigma,\\
\theta_{i+1}\sigma &= (\mathbf{M}[\![\mathtt{true}]\!]\sigma = tt) \;\rightarrow\; \mathbf{N}[\![\mathtt{skip}]\!]\theta_i\sigma,\ \mathbf{N}[\![\,\mathtt{x{:=}0}\,]\!]\theta\sigma\ =\\
&= \mathbf{N}[\![\mathtt{skip}]\!]\theta_i\sigma\\
&= \theta_i\sigma.
\end{aligned}$$

This means that all approximations θ_i are equal to $\mathbf{N}[\![\,\mathtt{diverge}\,]\!]\,\{\mathbf{N}[\![\,\mathtt{x{:=}0}\,]\!]\theta\}$ and thus for the limit we have

$$\mathbf{N}[\![\,\mathtt{while\ true\ do\ skip}\,]\!]\,\{\mathbf{N}[\![\,\mathtt{x{:=}0}\,]\!]\theta\} \;=\; \mathbf{N}[\![\,\mathtt{diverge}\,]\!]\,\{\mathbf{N}[\![\,\mathtt{x{:=}0}\,]\!]\theta\}.$$

Now $\mathbf{N}[\![\texttt{diverge}]\!]\theta'\sigma$ yields $\perp$ irrespective of θ' and σ, that is irrespective of the fact that the continuation in this case equals $\mathbf{N}[\![\,\texttt{x:=0}\,]\!]\theta$. The conclusion is therefore that indeed the bottom state $\perp$ is not passed to the next statement, and this is the reason that continuations do not need to be strict transformations.

Adding error exits to the language is easy in this framework. The clause on the assignment statement is changed: if evaluation of t yields an error then the continuation θ does not matter any more, the final answer of the computation will be ERR. If evaluation of t does not lead to an error, then the continuation has to be applied to the state resulting from executing the assignment:

$$\mathbf{N}[\![\,\texttt{x:=t}\,]\!]\theta\sigma \;=\; (\mathbf{M}[\![\texttt{t}]\!]\sigma = \text{ERROR}) \;\rightarrow\; \text{ERR},\; \theta(\sigma[\mathbf{M}[\![\texttt{t}]\!]\sigma/\texttt{x}])$$

So it appears that the strength of working with continuations is that you can ignore them: if in evaluating a statement all goes well then pass the resulting state to the continuation, if not (because of nontermination or an error) then halting the execution is mirrored by simply not using the continuation.

I will close this section with a few words on the denotational semantics of the **goto** statement. The difficulty there is to find out what the denotation for a label L must be. Intuitively a label corresponds to a "point in the program text" and this can be nicely captured by a continuation, viz. the transformation defining the effect of executing the program starting from label L. So the meaning $\mathbf{N}[\![\,\textbf{goto}\ \texttt{L}\,]\!]\theta\sigma$ of the statement **goto** L in state σ with respect to continuation θ, is that θ is disregarded, and instead the denotation for L, i.e. the continuation corresponding to L will be applied to σ.

Of course some work has still to be done before a full semantics is obtained, in particular a mechanism has to be given which can be used to build continuations from label definitions in program texts. Furthermore $\mathbf{N}[\![\texttt{S}]\!]$ must have an additional argument besides the state σ and the continuation θ, which serves to define the continuations corresponding to the labels occurring in the **goto** statements inside S. All this will be worked out in chapter 4, section 4 of this thesis.

1.2.7. A FEW THEOREMS

Having defined a few semantics for WHILE, a natural question to ask is whether these are equivalent. In this section I will first discuss some of these equivalence theorems and then I shall give a few remarks that are more technically oriented, a discussion of some of the proof techniques which will be used in the next chapters.

The first theorem is not an equivalence theorem however, it states a property of the continuation semantics for WHILE which is related to what is called "continuation removal" in the litterature [Sto77, MiS76]. In essence it states that in the formula $\mathbf{N}[\![\texttt{S}]\!]\theta\sigma$ the continuation θ will eventually be applied, that is WHILE is "jump free". The idea is that evaluation of S alone in σ will yield an intermediate state σ', and the final answer $\delta = \mathbf{N}[\![\texttt{S}]\!]\theta\sigma$ can be obtained by applying θ to this σ'.

I must be careful though because it is possible that $\mathtt{S}$ does not terminate, and in that case it is not possible to find a σ' to which θ can be applied,—remember θ is a function from Σ to A, and in the previous sections I did not allow $\perp \in \Sigma$. I will solve this by letting σ' range over $\Sigma_\perp$ and introducing the strict extension θ^* of θ; $\theta^* \colon \Sigma_\perp \to A$ is defined by $\theta^* \perp = \perp$ and $\theta^* \sigma = \theta\sigma$ for $\sigma \in \Sigma$. For a nonterminating evaluation of $\mathtt{S}$ in σ I then apply θ^* to $\perp$. This leads to the following theorem

Theorem 1. *For all* $\mathtt{S}$ *and all* $\sigma \in \Sigma$ *there is a* $\sigma' \in \Sigma_\perp$ *such that for all* θ *we have* $\mathbf{N}[\![\mathtt{S}]\!]\theta\sigma = \theta^*\sigma'$.

There is an immediate corollary to this theorem. If I substitute $\theta = \iota$, the identity continuation, defined by $\iota\sigma = \sigma$ for all $\sigma \in \Sigma$, then I get $\mathbf{N}[\![\mathtt{S}]\!]\iota\sigma = \iota^*\sigma'$. Now $\iota^*\sigma' = \sigma'$, and this means that the intermediate state σ' is in fact equal to $\mathbf{N}[\![\mathtt{S}]\!]\iota\sigma$. This result can also be justified in another way. The state $\sigma' \in \Sigma_\perp$ occurring in the theorem is the state to which θ^* has to be applied. This state must then be the result of evaluating $\mathtt{S}$ in σ. But then this state must be the same as the answer resulting from $\mathbf{N}[\![\mathtt{S}]\!]\iota\sigma$, because the continuation ι in this formula indicates that once $\mathtt{S}$ has been evaluated nothing more will be done.

Corollary 1. *For all* $\mathtt{S}$ *and all* $\sigma \in \Sigma$, $\mathbf{N}[\![\mathtt{S}]\!]\theta\sigma = \theta^*(\mathbf{N}[\![\mathtt{S}]\!]\iota\sigma)$.

Corollary 1 suggests that we can do without continuations for the language WHILE: because in $\mathbf{N}[\![\mathtt{S}]\!]\theta\sigma$ the continuation θ will always eventually be applied to the intermediate state $\mathbf{N}[\![\mathtt{S}]\!]\iota\sigma$, we have obtained a direct semantics again: $\mathbf{N}[\![\mathtt{S}]\!]\iota$ is the state transformation mapping an initial state σ to the state $\sigma' \in \Sigma_\perp$, resulting from evaluating $\mathtt{S}$ only. I showed earlier that for a direct semantics to be feasible the transformations defined by it must be strict. All this suggests that a direct semantics for WHILE is obtained by mapping the statement $\mathtt{S}$ onto $\{\mathbf{N}[\![\mathtt{S}]\!]\iota\}^*$ But then this semantics must be equivalent to the direct semantics defined in section 1.2.2.

Theorem 2. *For all* $\mathtt{S}$ *we have* $\mathbf{M}[\![\mathtt{S}]\!] = \{\mathbf{N}[\![\mathtt{S}]\!]\iota\}^*$.

There is a natural corollary to corollary 1 and theorem 2:

Corollary 2. *For all* $\mathtt{S}$, $\sigma \in \Sigma$ *and* $\sigma' \in \Sigma_\perp$ *we have that* $\mathbf{M}[\![\mathtt{S}]\!]\sigma = \sigma'$ *if and only if* $\mathbf{N}[\![\mathtt{S}]\!]\theta\sigma = \theta^*\sigma'$ *for all* θ.

The next theorem states that direct semantics and Cook semantics are equivalent. In essence it says that the last element of the trace obtained by applying the Cook valuation $\mathbf{C}$ to some $\mathtt{S}$ and σ must be equal to the result of the direct semantic function $\mathbf{M}$ applied to the same arguments. Here too there is a little complication because in section 1.2.2 I worked with $\perp$, and for the Cook semantics this was not the case.

Theorem 3. *For all* $\mathtt{S}$ *and* $\sigma \in \Sigma$ *we have that* $\mathbf{C}[\![\mathtt{S}]\!]\sigma$ *is finite iff* $\mathbf{M}[\![\mathtt{S}]\!]\sigma \neq \perp$, *and in that case also* $\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma) = \mathbf{M}[\![\mathtt{S}]\!]\sigma$ *holds.*

Notice that the fact that $\mathbf{M}[\![\mathtt{S}]\!]\sigma$ must be equal to $\perp$ if $\mathbf{C}[\![\mathtt{S}]\!]\sigma$ yields an infinite trace is only natural because an infinite trace can only be generated by a nonterminating computation.

It is easier to prove theorem 3 than a similar theorem relating continuation semantics and Cook semantics. This is so, because both direct and Cook semantics yield an outcome from which the state resulting from evaluation of a statement can easily be retrieved. However the equivalence of **N** and **C** now follows easily as a corollary to theorems 2 and 3.

Corollary 3. *For all* S *and* $\sigma \in \Sigma$ *we have that* $\mathbf{C}[\![\mathtt{S}]\!]\sigma$ *is finite iff* $\mathbf{N}[\![\mathtt{S}]\!]\iota\sigma \neq \bot$, *and in that case also* $\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma) = \mathbf{N}[\![\mathtt{S}]\!]\iota\sigma$ *holds.*

The last theorem to be established in this section is that the operational semantics of section 1.2.5 is equivalent to the other ones. Some care has to be exercised in formulating such a theorem. The problem is that the objects occurring in the operational semantics are finite, these are meant to be representations of denotations, the more abstract objects which are manipulated in denotational semantics.

So I need to establish a correspondence between these two worlds. This correspondence will be denoted by $\sim$, and the theorem will then be something like "if $s \sim \sigma$ then $\mathbf{O}[\![\mathtt{S}]\!]s \sim \mathbf{M}[\![\mathtt{S}]\!]\sigma$" (or $\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma)$, or $\mathbf{N}[\![\mathtt{S}]\!]\iota\sigma$, whichever semantics suits us the best). First of all I will define what it means for an s and a σ to be related through $\sim$.

A natural way to define $s \sim \sigma$ would be to demand that s and σ yield the same value for all x from **Var**. There are two reasons why this is not feasible. First, the value RETRIEVE x s of x in s will not be an element of V. Though in section 1.2.5 I did not specify the internal structure of an s in **Store**, I mentioned there that the operational semantics operates on representations of abstract values and not on these values themselves. So RETRIEVE x s will be a representation of a value in V, and one cannot demand more than that it will be a representation of $\sigma[\![\mathrm{x}]\!]$. For ease of notation I introduce a "derepresentation function" D which maps an operational value to the value of V of which it is a representation. All this would lead to the definition: $s \sim \sigma$ iff for all x in **Var** we have $D(\text{RETRIEVE x } s) = \sigma[\![\mathrm{x}]\!]$

However this does not work yet. A state σ is an infinite object which defines a value for all variables in **Var**, but an operational state s is finite and can therefore establish values of only a finite number of variables. So I cannot do more than define correspondence of s and σ only with respect to a finite subset of **Var**:

$s \sim \sigma$ w.r.t. A iff for all x in A: $D(\text{RETRIEVE x } s) = \sigma[\![\mathrm{x}]\!]$.

This finiteness is not a real restriction as far as our equivalence theorem is concerned, because evaluation of a statement S depends on and affects only the variables occurring in S. This property can easily be proven for all semantics developed in this chapter. Furthermore, the state s had better be defined for all variables in S, because otherwise $\mathbf{O}[\![\mathtt{S}]\!]s$ might be undefined (evaluation of S could need the value of a variable x which does not occur in s). If all variables occurring in S are defined in s, then $\mathbf{O}[\![\mathtt{S}]\!]s$ can only become undefined because of nontermination, for we have the following property: if RETRIEVE x s is defined for all variables in S and T, then we have that $\text{STEP}^i(\mathtt{S}, s, T)$ is defined for all i, where STEP^i stands for applying STEP i times.

The equivalence theorem can now be formulated

Theorem 4. *Let A be the set of all variables occurring in* S. *If $s \sim \sigma$ w.r.t. A, then* $\mathbf{O}[\![\mathtt{S}]\!]s$ *is defined iff* $\mathbf{N}[\![\mathtt{S}]\!]\iota\sigma \neq \bot$, *and in that case* $\mathbf{O}[\![\mathtt{S}]\!]s \sim \mathbf{N}[\![\mathtt{S}]\!]\iota\sigma$ *w.r.t. A.*

The rest of this section shall be devoted to a discussion on how the above theorems can be proved. It will be more technical than the presentation up till now. Readers who are not interested in technicalities can safely skip this as the material presented in later sections does not depend on it.

The most useful proof technique is structural induction: if I can prove that the theorem holds for the assignments, the atomic building blocks of **Stat** and furthermore, if I can prove that the theorem holds for a composite statement using the (induction) hypothesis that the theorem is true for the constituent statements, then the theorem must hold for all statements. For, each statement S can be broken into assignment statements, the theorem holds for all these statements, and using the proofs in the induction argument, a proof for S can be constructed (cf. the proof of strictness in section 1.2.2).

In such an inductive proof, the case for the **while** statement is in general the most complicated one. In order to prove that the theorem holds for the statement **while** b **do** S, the fact can be used that the theorem must be true for S,—this is the induction hypothesis. But often this is not enough. A technique that is useful in such a case is to prove the theorem for all approximations $\phi_0, \phi_1, \ldots, \phi_k, \ldots$ of $\mathbf{M}[\![$ **while** b **do** S $]\!]$, which can often be done by induction on k. The effect is that you get an induction argument inside another one. This could also be observed in the strictness proof in section 1.2.2.

Such an induction will only work if the property to be proven is such that

a) it does not only hold for the limit $\mathbf{M}[\![$ **while** b **do** S $]\!]$ but for all elements $\phi_0, \phi_1, \ldots$ of the approximation chain as well.

b) the property can be "pushed over the limit", it must be a property which can be proven to be true for the limit from the fact that it holds for all approximations.

The properties mentioned in theorems 1 and 2 are feasible in this respect. For instance, for theorem 2 we have: let $\phi_0, \phi_1, \ldots$ be the chain approximating $\phi = \mathbf{M}[\![\mathtt{S}]\!]$, and $\theta_0, \theta_1, \ldots$ the chain approximating $\theta = \mathbf{N}[\![\mathtt{S}]\!]\iota$, then

a) for all i: $\phi_i = (\theta_i)^*$.

b) if for all i $\phi_i = (\theta_i)^*$ then $\phi = \theta^*$ also holds for the limits.

The property stated in theorem 3 cannot be proved this way, because this property does not hold for all approximations. Take for instance the statement **while true do** x:=x. Approximations of $\mathbf{C}[\![$ **while true do** x:=x $]\!]\sigma$ are:

$\tau_0 = \langle\rangle$

$\tau_1 = \langle\sigma, \sigma\rangle$

$\tau_2 = \langle\sigma, \sigma, \sigma, \sigma\rangle$

etc, which means that for almost all approximations τ_i I get $\kappa(\tau_i) = \sigma$. On the other hand for all approximations of $\mathbf{M}[\![$ **while true do** x:=x $]\!]\sigma$ I get $\phi_i\sigma = \bot$.

Theorem 3 can be proven using another idea, namely to take only finite computations into account. In order to prove $\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma) = \mathbf{M}[\![\mathtt{S}]\!]\sigma$, I first investigate the case where $\mathbf{C}[\![\mathtt{S}]\!]\sigma$ is a finite sequence, i.e. corresponds to a finite computation. Then, using only the clauses of the definition of $\mathbf{C}$ (that is, the fact that the left hand sides are equal to the corresponding right hand sides), and not the fact that $\mathbf{C}[\![$ `while b do S` $]\!]$ is obtained as a limit, I can prove the theorem by induction on the length of $\mathbf{C}[\![\mathtt{S}]\!]\sigma$. The only case that remains to be proved is that if $\mathbf{C}[\![\mathtt{S}]\!]\sigma$ is infinite, that then $\mathbf{M}[\![\mathtt{S}]\!]\sigma = \perp$. This is hard, there is no suitable induction here. The best way to proceed is to show that if $\mathbf{M}[\![\mathtt{S}]\!]\sigma = \sigma' \neq \perp$, that then $\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma) = \sigma'$, that is again to consider only finite computations. This fact can be proved using structural induction, and in case of the `while` statement, by induction on the number of approximations needed.

In this respect it is worthwile to consider again the construction of the trace $\mathbf{C}[\![$ `x:=1; while x>0 do x:=x-1` $]\!]\sigma$ discussed in section 1.2.4. I use the idea of body replacement there, in each step I isolate a piece of the formula to be expanded which has the form of the left hand side of one of the clauses in the definition of $\mathbf{C}$, and replace it by the corresponding right hand side. In this construction I use only the clauses of $\mathbf{C}$ and not the limit construction for the `while` statement. This information is apparently sufficient to construct this trace corresponding to a finite computation.

Behind the proof sketched above lies a generalisation of this: if evaluation of `S` in σ does terminate, then $\mathbf{C}[\![\mathtt{S}]\!]\sigma$ as well as $\mathbf{M}[\![\mathtt{S}]\!]\sigma$ can be obtained by performing a finite number of body replacement steps using the clauses of $\mathbf{C}$, resp. $\mathbf{M}$. In other words, in order to obtain the result of a finite computation, the only properties of $\mathbf{C}$ and $\mathbf{M}$ needed are that the left hand sides in the clauses for $\mathbf{C}$ and $\mathbf{M}$ are equal to the corresponding right hand sides. The fact that the functions $\mathbf{C}[\![\mathtt{S}]\!]$ and $\mathbf{M}[\![\mathtt{S}]\!]$ are obtained as limits of approximation sequences is relevant only for infinite computations.

In chapter 4, section 4 I will prove a theorem (Lemma 4.3) similar to corollary 2 concerning the direct semantics for `goto` statements given by the function $\mathbf{A}$. The proof to be given there will be simpler than the one of theorem 2 on one hand, because I will not have to deal with nontermination there. On the other hand the situation is more complicated there, because the semantics defined through $\mathbf{A}$ has to deal with labels and jumps. This is also the reason why the theorem will be phrased like the above corollary instead of like theorem 2.

In chapters 3 and 4 of this thesis I will prove theorems like corollary 3. There I must give a direct proof, I will not be able to prove the analogue of theorem 3 first, because I will not have a suitable direct semantics at my disposal. In such a direct proof there are some complications, related to the choice of the induction hypothesis. A first possibility is to use the assertion from the theorem: $\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma) = \mathbf{N}[\![\mathtt{S}]\!]\iota\sigma$. However, then a proof for the case $\mathtt{S} \equiv \mathtt{S}_1;\mathtt{S}_2$ is not straightforward, because the right hand side rewrites to $\mathbf{N}[\![\mathtt{S}_1]\!]\{\mathbf{N}[\![\mathtt{S}_2]\!]\iota\}\sigma$ and in this formula the continuation does not have the required form. Only if $\mathtt{S}_1$ is an assignment $\mathtt{x}:=\mathtt{t}$ then $\mathbf{N}[\![\mathtt{S}_1;\mathtt{S}_2]\!]\iota\sigma$ can be brought into an acceptable form: $\mathbf{N}[\![\mathtt{x}:=\mathtt{t};\ \mathtt{S}_2]\!]\iota\sigma = \mathbf{N}[\![\mathtt{x}:=\mathtt{t}]\!]\{\mathbf{N}[\![\mathtt{S}_2]\!]\iota\}\sigma = \mathbf{N}[\![\mathtt{S}_2]\!]\iota(\sigma[\mathbf{M}[\![\mathtt{t}]\!]\sigma/\mathtt{x}])$.

In the equivalence proof in chapter 4, section 5, I take a way out by using a more involved induction than just structural induction. The idea is that $\mathtt{S}_1;\mathtt{S}_2$ must be rewritten by breaking $\mathtt{S}_1$ into parts until the first statement is an assignment, e.g. $((\mathtt{x}{:=}\mathtt{t};\ \mathtt{S}');\ \mathtt{S}'');\ \mathtt{S}_2$ is rewritten to $(\mathtt{x}{:=}\mathtt{t};\ \mathtt{S}');\ (\mathtt{S}'';\ \mathtt{S}_2)$ which is rewritten to $\mathtt{x}{:=}\mathtt{t};\ (\mathtt{S}';\ (\mathtt{S}'';\ \mathtt{S}_2))$. In order to make the induction work I have to introduce a new measure of the complexity of a statement different from the usual one: in a composition $\mathtt{S}_1;\mathtt{S}_2$ the complexity of the first part has to count more heavily than the complexity of the second part. The case $\mathtt{S} \equiv \mathtt{S}_1;\mathtt{S}_2$ can now be divided into subcases which can be proven by induction on this new complexity measure. For instance if $\mathtt{S}_1 \equiv \mathtt{S}';\mathtt{S}''$ then I get $\mathbf{N}[\![\,\mathtt{S}_1;\mathtt{S}_2\,]\!]\iota\sigma = \mathbf{N}[\![\,(\mathtt{S}';\mathtt{S}'');\mathtt{S}_2\,]\!]\iota\sigma = \mathbf{N}[\![\,\mathtt{S}';(\mathtt{S}'';\mathtt{S}_2)\,]\!]\iota\sigma$, where this last equality can easily be proved from the definition of $\mathbf{N}$. Now the complexity of the statement occurring in the right hand side is smaller than that of the original formula so the induction hypothesis can be used. All this has been worked out in the proof of Lemma 5.3.

Another approach must be taken for the pattern matcher in SNOBOL4 discussed in chapter 3. The strategy of rewriting the statement will not be feasible there. This is so because not all syntactic operators used to construct a composite statement in SNOBOL4 are distributive, e.g. SNOBOL4 has operators & and $\vee$ for constructing patterns $\mathtt{S}$ for which $(\mathtt{S}_1\ \&\ \mathtt{S}_2) \vee \mathtt{S}_3$ is not equivalent to $(\mathtt{S}_1 \vee \mathtt{S}_3)\ \&\ (\mathtt{S}_2 \vee \mathtt{S}_3)$. In the proof there the induction hypothesis must be strengthened. With respect to the example of corollary 3 this would correspond to changing the hypothesis from $\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma) = \mathbf{N}[\![\mathtt{S}]\!]\iota\sigma$ into: for all θ we have that $\mathbf{N}[\![\mathtt{S}]\!]\theta\sigma = \theta(\kappa(\mathbf{C}[\![\mathtt{S}]\!]\sigma))$ Now induction can be used for the case $\mathtt{S} \equiv \mathtt{S}_1;\mathtt{S}_2$ because $\mathbf{N}[\![\,\mathtt{S}_1;\mathtt{S}_2\,]\!]\theta\sigma = \mathbf{N}[\![\mathtt{S}_1]\!]\,\{\mathbf{N}[\![\mathtt{S}_2]\!]\theta\}\,\sigma$, and by structural induction ($\mathtt{S}_1$ is simpler than $\mathtt{S}_1;\mathtt{S}_2$) this is equal to $\{\mathbf{N}[\![\mathtt{S}_2]\!]\theta\}\sigma'$, where $\sigma' = \kappa(\mathbf{C}[\![\mathtt{S}_1]\!]\sigma)$. Structural induction again (now because $\mathtt{S}_2$ is less complicated than $\mathtt{S}_1;\mathtt{S}_2$) yields that this equals $\theta(\kappa(\mathbf{C}[\![\mathtt{S}_2]\!]\sigma'))$ and this is equal to $\theta(\kappa(\mathbf{C}[\![\mathtt{S}_1;\mathtt{S}_2]\!]\sigma))$ by definition of $\mathbf{C}$.

In theorem 4 I decided to choose continuation semantics to compare with the operational semantics because the proof will run more smoothly. The proof will again be inductive, and it will be similar to the proof of theorem 3. Simple structural induction will not work here for the same reason that it did not work in the proof of theorem 3,—the semantics are too dissimilar. The proof consists of two parts.

The first part states that if $s \sim \sigma$ w.r.t. A and $\mathbf{O}[\![\mathtt{S}]\!]s$ is defined then $\mathbf{N}[\![\mathtt{S}]\!]\iota\sigma \neq \perp$ and $\mathbf{O}[\![\mathtt{S}]\!]s \sim \mathbf{N}[\![\mathtt{S}]\!]\iota\sigma$ w.r.t. A. This proof is by induction on the number of STEPs needed in evaluating $\mathbf{O}[\![\mathtt{S}]\!]s$ However, as it stands induction does not work, the induction hypothesis has to be made stronger. This is because $\mathbf{O}[\![\mathtt{S}]\!]s$ is defined through $\mathrm{STEP}^i(\mathtt{S}, s, \langle\,\rangle)$ and in the proof one needs to apply some induction hypothesis to the intermediate machine configurations $(\mathtt{S}, s, T)$ which will be the result of applying some STEPs to the initial configuration. This induction hypothesis should state some properties of the final state resulting from evaluating $(\mathtt{S}, s, T)$ through a finite number of STEPs.

Now I can explain the reason why I choose to prove $\mathbf{O}$ equivalent to the function

$\mathbf{N}$ of continuation semantics: in order to establish a correspondence between the configuration $(\mathtt{S}, s, T)$ and an intermediate stage of evaluation of $\mathtt{S}$ according to some denotational semantics, this denotational semantics should have some counterpart of the "syntactic continuation" T. Using $\mathbf{N}$ the induction hypothesis can be something like: if $T \sim \theta$, and $s \sim \sigma$ w.r.t. A and if $\mathbf{P}(\mathtt{S}, s, T)$ is defined, then $\mathbf{P}(\mathtt{S}, s, T) \sim \mathbf{N}[\![\mathtt{S}]\!]\theta\sigma$ w.r.t. A. Here $\mathbf{P}(\mathtt{S}, s, T)$ is the store in the final machine configuration $\mathrm{STEP}^i(\mathtt{S}, s, T)$ for i big enough.

So I must establish a suitable correspondence relation $\sim$ between T's and θ's which must have the property that $\langle\,\rangle \sim \iota$, so that the theorem can be obtained from the induction hypothesis. Again a derepresentation function D works, defined by

$D\langle\,\rangle = \iota$

$D(\langle \mathtt{S}\rangle {}^\wedge T) = \mathbf{N}[\![\mathtt{S}]\!](D\,T)$.

This definition is a natural one: the effect of $T = \langle\,\rangle$ in $(\mathtt{S}, s, T)$ is that after evaluation of $\mathtt{S}$ the overall execution is finished, and this corresponds to the identity continuation ι; secondly, the effect of $T = \langle \mathtt{S}'\rangle {}^\wedge T'$ is that after evaluation of $\mathtt{S}$, first of all $\mathtt{S}'$ has to be executed, followed by a computation defined through T', and this corresponds (inductively) to the continuation $\mathbf{N}[\![\mathtt{S}']\!](D\,T')$.

So the first half of the theorem can be proved using the induction hypothesis: if $s \sim \sigma$ w.r.t. A and $\mathbf{P}(\mathtt{S}, s, T)$ is defined, then $\mathbf{N}[\![\mathtt{S}]\!](D\,T)\sigma \neq \bot$ and $\mathbf{P}(\mathtt{S}, s, T) \sim \mathbf{N}[\![\mathtt{S}]\!](D\,T)\sigma$ w.r.t. A. The proof of this is straightforward by induction on the number of STEPs needed to obtain $\mathbf{P}(\mathtt{S}, s, T)$. It works out smoothly because STEP decomposes the statement to be evaluated in the same manner as can be observed in the clauses of $\mathbf{N}$. Again, the only property of $\mathbf{N}$ needed here is equality of the left hand sides to their right hand sides in the clauses defining $\mathbf{N}$.

This ends the discussion of the first part of the proof of the theorem. The other half of the theorem claims: if $s \sim \sigma$ w.r.t. A and $\mathbf{N}[\![\mathtt{S}]\!]\iota\sigma \neq \bot$, then $\mathbf{O}[\![\mathtt{S}]\!]s$ is defined, and $\mathbf{O}[\![\mathtt{S}]\!]s \sim \mathbf{N}[\![\mathtt{S}]\!]\iota\sigma$ w.r.t. A. To prove this we use the following induction pattern, which is in fact a combination of this claim and theorem 1: if $\mathbf{N}[\![\mathtt{S}]\!]\theta\sigma = \sigma' \in \Sigma$, and if $s \sim \sigma$ w.r.t. A, then there exists $\sigma'' \in \Sigma$, s'' and i such that $s'' \sim \sigma''$ w.r.t. A, $\sigma' = \theta\sigma''$ and for all T we have $\mathrm{STEP}^i(\mathtt{S}, s, T) = (\mathtt{skip}, s'', T)$.

Theorem 1 stated that in $\mathbf{N}[\![\mathtt{S}]\!]\theta\sigma$ the continuation θ will eventually be applied to some intermediate state σ''. The above lemma states that a related property must also hold for our operational semantics: in the corresponding evaluation of $\mathbf{P}(\mathtt{S}, s, T)$, the syntactic continuation will eventually be used in an intermediate store s'' corresponding to σ''.

This lemma can be proved with structural induction and, for the `while` case, induction on the number of approximations needed. The second half of the theorem follows from this lemma by choosing $\theta = \iota$ and $T = \langle\,\rangle$. This concludes the description of the proof of theorem 4.

In chapter 3 I will prove an equivalence similar to theorem 4 for the pattern matcher in SNOBOL4. The proof there will be more involved mainly because the language considered there is more complicated. On the other hand the SNOBOL4 fragment defined there will be such that pattern matching always terminates. This will be proved in chapter 3, lemma's 5.9–12. This means that the counterpart of $\mathbf{O}[\![\mathbf{S}]\!]s$ will always be defined there, and therefore it will be sufficient to prove only the easy part of the theorem. The proof given there will correspond to the first part of the proof described above.

1.3. Using continuations

The final section of this chapter deals with my experiences with continuations in the research that led to the papers reprinted in the next chapters. Denotational semantics, and therefore continuations can be applied in several ways. First of all continuations can be used to provide a semantics for an existing language, or for a language set up to study certain programming concepts like jumps, coroutines and the like. Which of such concepts can better be defined through direct semantics and for which is continuation semantics superior? Questions like these will be dealt with in section 1.3.1.

Having devised a semantics for a language one then has a standard at one's disposal which can be used to reason about the language or about systems of which the language is a part, e.g. an implementation of the language or a formal proof system for it. In section 1.3.2 a few remarks will be made on these topics, and again I will investigate which style of semantics is better suited for these purposes, direct or continuation semantics.

1.3.1. CONTINUATIONS AND SEMANTICS

In the first years after the main ideas of denotational semantics had been introduced, this theory was used to define the semantics of full, existing pogramming languages, like LISP [Gor73], SNOBOL4 [Ten73], Algol68 [Mil74], Algol60 [Mos74], SAL [MiS76], Gedanken [Ten76] and Pascal [Ten77].

All of these semantics use continuations, the idea was that one cannot go without these if all kinds of control structures have to be modelled that do not adhere to the single-entry single-exit principle: "...if one wishes to deal with realistic programming language features, such as input/output or complicated control structures, one must proceed almost immediately to continuation semantics..." [Wan82].

However such ideas are to some extent contradicted by the work of the VDM group (Vienna Development Method) [BjJ78]. They provided direct denotational semantics for languages like PL/I [BBH74], Algol60 [HeJ78], CHILL [HaB81] and work on Ada is going on [BjO80]. So, at least for languages containing **goto** statements and labels, even if these labels can be passed as actual parameters like in Algol60, direct semantics is possible.

Jones [Jon78] and Bjørner [Bjø80] claim that direct semantics has some advantages over continuation semantics, the main ones being that direct semantics is intuitively more appealing (continuations describe the flow of control in a more indirect way, which makes them harder to understand), and that their direct semantics can be "implemented" in a language that is machine processable (I will return to that in section 1.3.2). On the other hand, they also remark that continuations are more flexible in that there are programming language concepts, like coroutines, that can be modelled with continuations but not with their version of direct semantics.

In this section I will investigate the pros and the cons of continuation semantics versus direct semantics and I will do this by studying direct as well as continuation semantics for some of the programming concepts to be treated in chapters 2 through 5. First (section 1.3.1.1) I will introduce some of the ideas from the VDM group by studying the **goto** statement. After that (1.3.1.2) I will discuss semantics for backtracking, introducing some of the ideas to be worked out in chapter 3. Finally I will give some remarks on the semantics of some of the concepts to be treated in chapter 5, namely nonterminating processes, input/output, (a restricted form of) parallellism and dynamic process creation.

1.3.1.1. Labels and goto statements. Chapter 4 of this thesis deals with a Hoare like proof system for a language containing labels and **goto** statements. In order to justify this system I define a semantics there for these concepts that uses continuations. At the end of section 1.2.6 I have given a few remarks on this semantics and there is no need to discuss it further here. Instead I will now give a few comments on the meaning function **A** which is also introduced in chapter 4, cf. the end of section 4 of that chapter. This function **A** establishes a direct semantics for the **goto** statement.

The technique to be applied is similar to the one behind the direct semantics for error exits presented in section 1.2.6, namely to code additional information into the intermediate states. In this case to a state an optional second component can be added, namely the name of a label. The idea is that a statement that contains a **goto** can terminate in two ways. First it can terminate normally, by "reaching the right hand end of the statement" so to speak. The other possibility is that evaluation can be broken off through execution of a substatement that is a **goto**, that is by "jumping away" to some label. In the first case the meaning function **A** will yield a regular state, in the second case a state-label pair will be delivered, consisting of the state in which the **goto** was encountered and the label occurring in this **goto** statement. The function **A** has therefore the following functionality:

$$\mathbf{A}\colon \mathbf{Stat} \rightarrow \Sigma_\perp \rightarrow (\Sigma_\perp \cup \Sigma \times \mathbf{Lvar})$$

Here **Lvar** is the set of the label names. Notice that $\perp$ has to be added again to the states, because we are discussing direct semantics here. One might say that statements with **goto**'s occurring in it are single-entry multi-exit statements, there is one normal exit and possibly many **goto**-exits.

The meaning of the statement **goto** L can now easily be given:

$\mathbf{A}[\![\, \mathbf{goto}\; \mathrm{L} \,]\!]\perp = \perp$

$\mathbf{A}[\![\, \mathbf{goto}\; \mathrm{L} \,]\!]\sigma = \langle \sigma, \mathrm{L} \rangle, \qquad \text{for } \sigma \neq \perp$

For a statement of the form **goto** L; S we must have

$$\mathbf{A}[\![\, \mathbf{goto}\ \mathtt{L};\ \mathtt{S}\,]\!]\sigma = \langle \sigma, \mathtt{L}\rangle$$

if $\sigma \neq \perp$. This can be realised by defining the meaning of composition more carefully:

$$\mathbf{A}[\![\, \mathtt{S}_1;\ \mathtt{S}_2\,]\!]\sigma = (\mathbf{A}[\![\mathtt{S}_1]\!]\sigma \in \Sigma_\perp) \ \rightarrow\ \mathbf{A}[\![\mathtt{S}_2]\!](\mathbf{A}[\![\mathtt{S}_1]\!]\sigma),\ \mathbf{A}[\![\mathtt{S}_1]\!]\sigma$$

If there are **goto** statements in a language, there must be a way to define labels as well. To this end I introduce the class **Prog** of programs with elements P. Programs are just compound statements, sequences $\mathtt{L}_1\colon \mathtt{S}_1;\ \ldots\ ; \mathtt{L}_n\colon \mathtt{S}_n$ of labelled statements. In chapter 4 I do not define a direct semantics for programs because it is not needed there. Lemma 4.3 in that chapter describes a relation between the meaning function **A** and the meaning function **N** defining continuation semantics which gives sufficient information for my purposes.

The VDM approach however works with direct semantics only, so in that case a direct semantics has to be given for programs as well. The idea is to view a program $\mathtt{P} \equiv \mathtt{L}_1\colon \mathtt{S}_1;\ \ldots\ ; \mathtt{L}_n\colon \mathtt{S}_n$ not only as a multi-exit statement (there can be substatements **goto** L in P where $\mathtt{L} \not\equiv \mathtt{L}_i$ for all i), but also as a multi-entry statement. The program P has n entry points characterised by the labels $\mathtt{L}_i$. Therefore the meaning $\mathbf{A}[\![\mathtt{P}]\!]$ will not be a function from $\Sigma_\perp$ to $(\Sigma_\perp \cup (\Sigma \times \mathbf{Lvar}))$, but instead we have

$$\mathbf{A}[\![\mathtt{P}]\!]\colon (\Sigma_\perp \times \mathbf{Lvar}) \ \rightarrow\ (\Sigma_\perp \cup (\Sigma \times \mathbf{Lvar})),$$

and we have the following definition:

$$\begin{aligned}
&\mathbf{A}[\![\, \mathtt{L}_1\colon \mathtt{S}_1;\ \ldots\ ; \mathtt{L}_n\colon \mathtt{S}_n\,]\!]\langle \sigma, \mathtt{L}\rangle \ = \\
&\quad \sigma = \perp\ \rightarrow\ \perp, \\
&\quad \mathtt{L} \not\in \{\mathtt{L}_1, \ldots, \mathtt{L}_n\}\ \rightarrow\ \langle \sigma, \mathtt{L}\rangle, \\
&\quad \mathtt{L} \equiv \mathtt{L}_i\ \rightarrow \\
&\qquad \Big(\mathbf{A}[\![\mathtt{S}_i]\!]\sigma \in (\Sigma \times \mathbf{Lvar})\ \rightarrow\ \mathbf{A}[\![\, \mathtt{L}_1\colon \mathtt{S}_1;\ \ldots\ ; \mathtt{L}_n\colon \mathtt{S}_n\,]\!](\mathbf{A}[\![\mathtt{S}_i]\!]\sigma), \\
&\qquad\ \mathbf{A}[\![\mathtt{S}_i]\!]\sigma \in \Sigma_\perp \text{ and } i \neq n\ \rightarrow \\
&\qquad\qquad \mathbf{A}[\![\, \mathtt{L}_1\colon \mathtt{S}_1;\ \ldots\ ; \mathtt{L}_n\colon \mathtt{S}_n\,]\!]\,\langle \mathbf{A}[\![\mathtt{S}_i]\!]\sigma,\ \mathtt{L}_{i+1}\rangle,\ \mathbf{A}[\![\mathtt{S}_i]\!]\sigma\Big)
\end{aligned}$$

An explanation. Let $\mathtt{P} \equiv \mathtt{L}_1\colon \mathtt{S}_1;\ \ldots\ ; \mathtt{L}_n\colon \mathtt{S}_n$. Now there are a few possiblities. If the argument of $\mathbf{A}[\![\mathtt{P}]\!]$ is the result of a nonterminating computation then $\perp$ should be passed. If the argument of $\mathbf{A}[\![\mathtt{P}]\!]$ specifies an entry point not in P, then the argument should be passed unaltered. The interesting case is when L is an $\mathtt{L}_i$. Then first of all $\mathtt{S}_i$ has to be executed in σ. If this terminates through a jump out of $\mathtt{S}_i$ then $\mathbf{A}[\![\mathtt{P}]\!]$ should again be applied to the outcome $\mathbf{A}[\![\mathtt{S}_i]\!]$ which is some $\langle \sigma', \mathtt{L}'\rangle$, because $\mathtt{L}'$ might be some $\mathtt{L}_j$. If evaluation of $\mathtt{S}_i$ terminates normally then $\mathbf{A}[\![\mathtt{S}_i]\!]$ is some σ' and $\mathtt{S}_{i+1}$ should now be executed in σ', which is equivalent to executing P in $\langle \sigma', \mathtt{L}_{i+1}\rangle$. This is not correct however if $i = n$, in that case the outcome $\mathbf{A}[\![\mathtt{S}_i]\!]$ is the final result of the whole computation of P.

Notice that this definition is circular. Again, the desired solution to the above equation can be obtained through approximation: the first approximation should now be the function that yields $\perp$ in all arguments of the form $\langle \sigma, \mathtt{L}_i\rangle$.

The language that I presented here is rather simple. However this style of establishing a semantics is quite powerful. For instance only a small extension to this scheme is needed for a semantics for full Algol60 [HeJ78].

A few conclusions. The main advantage of this approach is that it is more straightforward than continuation semantics, the idea behind this approach is easy to understand. However, a price has been paid. First of all, the definition of the meaning **A**[[P]] of a program P is hairy. Through introduction of a sensible notation this can be and has been straightened out somewhat, but in essence all cases and subcases must be discriminated. The other drawback is that all kinds of information have to be coded in the intermediate states, the definition of the state has to be expanded for every feature added to the language that does not conform to the single-entry single-exit principle. For instance, if one wants to add error exits to the above language, then the states have to be redefined again, which is not the case in the continuation semantics of chapter 4.

1.3.1.2. Backtracking. In order to study this concept I extend the language WHILE from section 1.2 adding two new statements, **try** and **fail**. Here is the new syntax:

$$\begin{aligned} \mathtt{S} \; ::= \;& \mathtt{x{:=}t} \mid \mathtt{S_1;S_2} \mid \mathbf{if}\ \mathtt{b}\ \mathbf{then}\ \mathtt{S_1}\ \mathbf{else}\ \mathtt{S_2} \mid \mathbf{while}\ \mathtt{b}\ \mathbf{do}\ \mathtt{S} \mid \\ & \mathbf{try}(\mathtt{S_1},\mathtt{S_2}) \mid \mathbf{fail} \end{aligned}$$

One idea behind backtracking is that evaluation of a statement may fail. In this language such a failure occurs whenever the statement **fail** is executed. Another aspect is that during execution of a program choices will be made between alternatives. Every time a **try** statement is executed, a new alternative is introduced. Executing **try**(S_1, S_2) has the same effect as executing S_1 but there is a side effect, namely that a new alternative is added to the set of alternatives already present, which will be pursued if S_1 terminates in failure.

If in the sequel of the computation a failure occurs (and no other **try** statement has been executed in the meantime), then a backtrack is performed which corresponds to a jump to the alternative in the **try** statement that has been evaluated most recently, which in our case is the statement **try**(S_1, S_2). Execution of the program now continues with evaluation of the second alternative S_2 followed by evaluation of the statements following the **try** statement. If during this evaluation another failure occurs then the statement **try**(S_1, S_2) does not provide further alternatives, and evaluation will backtrack to the most recently executed **try** statement which still has an open alternative.

In short, every time a **try** statement is evaluated, a new alternative is obtained (i.e. pushed on the stack of open alternatives), and every time a **fail** statement is evaluated backtracking occurs, which amounts to executing a jump to the alternative in the most recently executed **try** statement which has an open alternative (i.e. the alternative on top of the stack). If no such alternative exists then the whole computation ends in failure.

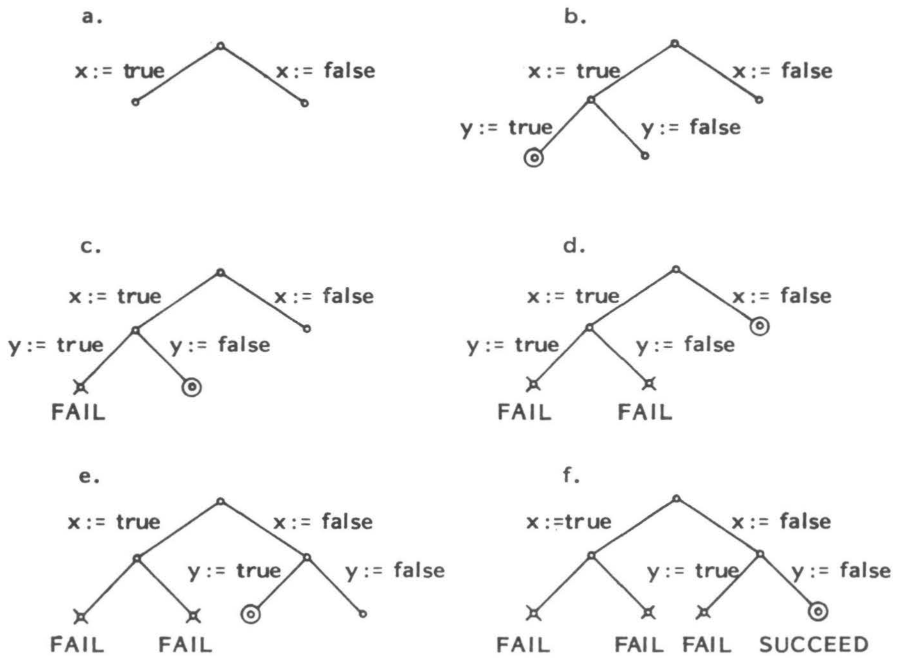

Figure 2. The tree of possibilities corresponding to a backtrack program. The circled node is the node where control resides.

A nice introduction into the backtracking style of programming is [Coh79]. By way of an example I present a little program that determines whether a boolean expression b in the boolean variables x and y is satisfiable. This will be done by trying all four possible pairs of values of x and y:

```
try(x:=true, x:=false);
try(y:=true, y:=false);
if not b then fail
```

If this program terminates in a non failure state then b is satisfiable otherwise it is not. To see this consider for instance the expression b ≡ **not** (x **or** y). I explore the "tree of possibilities" (cf. figure 2).

The first statement introduces an alternative, we first choose the option x:=**true**, leaving the possibility x:=**false** for later use, and arrive at the situation depicted in figure 2a. Again there is a choice, and we obtain a second statement to backtrack to (fig. 2b). After that the conditional statement is evaluated, which leads to the execution of the **fail** statement, and thus we arrive at figure 2c. The possibility y:=**false** also leads to failure, and the situation of figure 2d ensues. Now a new choice must be made, as again there are two possible assignments to y. First the "wrong one" is taken, giving fig. 2e, and finally the value of both x and y are such that the boolean expression in the if-statement evaluates to **false**, and the program succeeds (fig. 2f).

Notice that every evaluation of a try statement generates a new alternative. Therefore if a try occurs in the body of a loop, then in every iteration of this loop a new alternative will be created. Consider for instance the program:

```
y:=x; ready:=false;
while (y>0 and not ready) do
try(y:=y-1,
    y:=y+1; if y*y=x then ready:=true else fail);
if not ready then fail
```

If the initial value of x is a positive integer, then this program terminates in failure if x is not a square, otherwise it terminates in success with y equal to the square root of x. The idea is that in every iteration of the loop an alternative is created, so after x iterations the loop terminates with y=0 and a stack of x alternatives. Then all alternatives are tried, every alternative increases y by 1, so the effect is that for y = 1, ..., x it is checked whether y*y=x. If such a y is found then the search terminates because **ready** is set to **true**.

In chapter 3 I consider a fragment of SNOBOL4 in which the same concepts are present, in a style that is geared more towards the problem of string matching.

I now turn to the semantics of the language introduced here, and the first semantics will be continuation semantics. The most important observation to be made is that the meaning $\mathbf{N}[\![\mathtt{S}]\!]$ of a statement now depends on two continuations instead of one: after S has been worked through the computation can proceed in two directions depending on whether evaluation of S terminated in success or in failure.

Thus the meaning $\mathbf{N}[\![\mathtt{S}]\!]$ will be a function that takes as arguments, besides a state σ, a success continuation θ and a failure continuation β. The set of the success continuations will be called **Suc** and the failure continuations form the set **Alt**. The functionality of **N** is

$$\mathbf{N}\colon \mathbf{Stat} \to \mathbf{Suc} \to \mathbf{Alt} \to \Sigma \to A,$$

and $\mathbf{N}[\![\mathtt{S}]\!]\theta\beta\sigma$ denotes the answer obtained by evaluating S in σ, given that the computation will proceed according to θ if evaluation of S terminates in success, and according to β if we have termination in failure.

In the former case this evaluation of S does not only transform the initial state σ into a new state σ', but also, through execution of substatements which are try statements, new alternatives are created, that is the failure continuation β is updated as well yielding β'. The future of the computation modelled by θ depends on this σ' and β', and therefore θ must be a function taking as arguments a failure continuation and a state and yielding an answer: $\mathbf{Suc} = \mathbf{Alt} \to \Sigma \to A$.

Failure continuations are less complex, we have $\mathbf{Alt} = \Sigma \rightarrow A$. This is so, because β models how the computation will proceed if evaluation of S terminates in failure. Now the only effect of a thus terminating evaluation is that the state has been transformed, all new alternatives generated during execution of S must have been used up before evaluation of S as a whole can terminate in failure.

Now that the form of the domains has been determined the semantic clauses can be written down straightforwardly:

$$\begin{aligned}
&\mathbf{N}[\![\mathtt{x{:=}t}]\!]\theta\beta\sigma \;=\; \theta\beta(\sigma[\mathbf{M}[\![\mathtt{t}]\!]\sigma/\mathtt{x}])\\
&\mathbf{N}[\![\,\mathtt{S}_1;\mathtt{S}_2\,]\!]\theta\beta\sigma \;=\; \mathbf{N}[\![\mathtt{S}_1]\!]\,\{\mathbf{N}[\![\mathtt{S}_2]\!]\theta\}\,\beta\sigma\\
&\mathbf{N}[\![\,\mathbf{if}\ \mathtt{b}\ \mathbf{then}\ \mathtt{S}_1\ \mathbf{else}\ \mathtt{S}_2\,]\!]\theta\beta\sigma \;=\\
&\qquad(\mathbf{M}[\![\mathtt{b}]\!]\sigma \;=\; tt) \;\rightarrow\; \mathbf{N}[\![\mathtt{S}_1]\!]\theta\beta\sigma,\; \mathbf{N}[\![\mathtt{S}_2]\!]\theta\beta\sigma\\
&\mathbf{N}[\![\,\mathbf{while}\ \mathtt{b}\ \mathbf{do}\ \mathtt{S}\,]\!]\theta\beta\sigma \;=\\
&\qquad(\mathbf{M}[\![\mathtt{b}]\!]\sigma \;=\; tt) \;\rightarrow\; \mathbf{N}[\![\mathtt{S}]\!]\;\{\mathbf{N}[\![\,\mathbf{while}\ \mathtt{b}\ \mathbf{do}\ \mathtt{S}\,]\!]\theta\}\,\beta\sigma,\; \theta\beta\sigma\\
&\mathbf{N}[\![\,\mathbf{try}(\mathtt{S}_1,\mathtt{S}_2)\,]\!]\theta\beta\sigma \;=\; \mathbf{N}[\![\mathtt{S}_1]\!]\theta\,\{\mathbf{N}[\![\mathtt{S}_2]\!]\theta\beta\}\,\sigma\\
&\mathbf{N}[\![\,\mathbf{fail}\,]\!]\theta\beta\sigma \;=\; \beta\sigma
\end{aligned}$$

Next I turn to direct semantics, for which I will discuss two approaches. The first one has been used in work on the semantics of SNOBOL4 patterns [Gim73, Gim75]. The ideas developed there can be explained best for a variant of the language introduced above: now the **fail** statement does not only have the effect that a jump is executed to the last open alternative but also that the state is restored, i.e. if evaluation backtracks to **try**(S_1, S_2) then evalution resumes with execution of S_2 in the same state in which S_1 was evaluated. Thus the statement **try**(S_1, S_2) resembles the nondeterministic choice statement $S_1 \cup S_2$ [Bak80, ch.7]. The only difference is that **try**(S_1, S_2) specifies explicitly the order in which the alternatives have to be tried. The consequence is that, while the meaning $\mathbf{M}[\![\,S_1 \cup S_2\,]\!]\sigma$ can be a set consisting of all final states in $\mathbf{M}[\![S_1]\!]\sigma$ and $\mathbf{M}[\![\,S_2\,]\!]\sigma$, the meaning $\mathbf{M}[\![\,\mathbf{try}(S_1, S_2)\,]\!]\sigma$ must be a sequence: $\mathbf{M}[\![S_1]\!]\sigma \mathbin{^\wedge} \mathbf{M}[\![S_2]\!]\sigma$. Notice that, though in the end evaluation of a backtrack program will yield a single final state, for the intermediate stages the meaning of a statement must be a sequence. Not only the final state resulting of evaluation of a statement is needed, but also the results of the open alternatives, because otherwise a compositional definition cannot be given for the composition operator ;. In this setup, if we give a careful definition of the meaning of this operator, then we are able to obtain a full semantics for our language.

If one tries to model the language having the original semantics for **fail**, then the outcome $\mathbf{M}[\![S]\!]\sigma$ of evaluation of a statement S cannot longer be a sequence of states. Only the first element of the sequence can be a state, the other elements must be state transformations, because in **try**(S_1, S_2) the final state resulting from evaluation of S_2 after a failure depends on the state in which the computation failed. So we now get tuples of functions and much of the simplicity of the original approach is gone,—in fact we have a result that is quite close to continuation semantics.

Another approach to designing a direct semantics is to proceed along the VDM lines: handle **fail** statements similarly to the way in which VDM semantics handles jumps. There are two problems there: a failure corresponds to a jump with an anonymous target while the **goto** statement explicitly names its target, and the other problem is that this target is determined dynamically.

The first problem might be solved by changing the syntax so that every **try** statement has a tag which characterises its occurrence in the program or, in other words, all **try** statements must be labelled. The second problem can be handled by again extending the intermediate states, an extended state must now be an element of $(\Sigma_\perp \times \mathbf{Lvar}^*) \cup (\Sigma \times \mathbf{Lvar}^* \times \mathbf{Lvar})$. The optional third component of such an extended state has the same function as in the VDM-semantics of the **goto** statement: it indicates that the statement just executed has failed and provides the label of the **try** statement to which a backtrack has to be performed. The second component of an extended state is an element from the set $\mathbf{Lvar}^*$ which models the stack of open alternatives, it is the sequence of the labels of all **try** statements that have been encountered but not yet backtracked to.

Typical semantical clauses would now be (ς stands for an element from $\mathbf{Lvar}^*$):

$$\mathbf{M}[\![\,\mathtt{L : try(S_1, S_2)}\,]\!]\langle\sigma,\varsigma\rangle \;=\; \mathbf{M}[\![\mathtt{S_1}]\!]\langle\sigma,\ \langle\mathtt{L}\rangle^\wedge\varsigma\rangle$$

The effect of executing a **try** is to execute $\mathtt{S_1}$ and to add a new alternative to the stack ς.

Another clause: if ς is not empty then we must have:

$$\mathbf{M}[\![\,\mathtt{fail}\,]\!]\langle\sigma,\varsigma\rangle \;=\; \langle\sigma,\ \mathit{rest}\ \varsigma,\ \mathit{first}\ \varsigma\rangle$$

Failure generates an abnormal state, the third component of the resulting tuple specifies the **try** statement to which must be backtracked, the remaining alternatives are kept in the second component.

Again the complicated clause will be the one for the whole program. In this clause failure results of the substatements must be trapped and it must be specified how the backtracking is to be done. This clause must be set up with care because it is possible to backtrack e.g. from outside a **while** statement into its body. However this can be realised: a similar problem was encountered in the VDM semantics of Algol60, since in this language it is possible to jump into e.g. the **then**-part of a conditional statement from outside this statement.

The differences between the continuation and the direct approach show clearly here. Once the domains have been laid out correctly continuation semantics yields a regular and compact set of semantic equations, though it takes some thinking before the right domains are determined. On the other hand, direct semantics has a more baroque appearance: there must be extra information in the states and the clauses tend to be more complex, there are more cases to be distinguished.

1.3.1.3. Input, output and process creation. In this section I introduce some concepts from the language DNP which is discussed in chapter 5. To this end I extend the language WHILE in yet another direction, adding a **read** statement, a **write** statement and a **fork** statement. This addition makes WHILE a language in

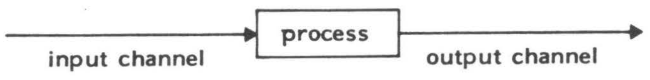

Figure 3. A process

which arrays of communicating processes can be defined. The syntax is

$$S ::= x:=t \mid S_1; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } b \textbf{ do } S \mid \texttt{write}(t) \mid \texttt{read}(x) \mid \texttt{fork}(x)$$

In the sequel a program in execution will be called a process. Each process has exactly one input channel and one output channel connected to it. Execution of the statement **write(t)** has the effect that the value of **t** is computed and written on the output channel, the effect of the statement **read(x)** is that a new value is read from the input channel which is then assigned to x. If there are no more values on the input channel then the process blocks (terminates). The values on the input channel are assumed to be put there a priori, so there is no need to model a process waiting for input. Communication is asynchronous.

A process can be modelled by a function which takes an input stream as an argument and yields an output stream as a result. The input stream is the sequence of all values assumed to be preloaded on the input channel, and the output stream is the sequence of all values to be written by the process on the output channel. Both streams can very well be infinite, and this means that nonterminating processes are meaningful in this setting. I give an example, a "2-filter" described by the program

```
while true do
begin read(x);
      if odd(x) then write(x)
end.
```

This programs filters all even numbers, passing only odd numbers from its input channel to its output channel. A process can be depicted as in figure 3.

The other new concept in the language is the **fork** construct, described by a statement of the form **fork(x)**. This statement can be regarded as a combination of the unix fork and the unix pipe. When a process executes the statement **fork(x)**, the effect is that an almost identical copy of the process is constructed. I call the original process the mother and the new process the daughter. After the **fork** has been evaluated both processes continue execution with the statement following the **fork**. There is no sharing of variables, each process has its own set of variables having the values they had in the mother process when the **fork** was executed.

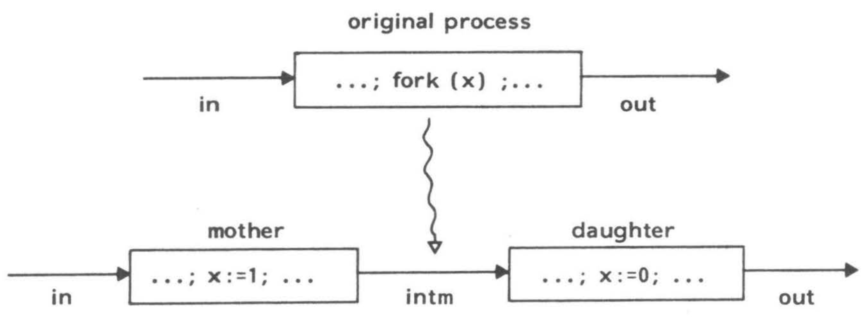

Figure 4. The effect of the `fork` statement.

There are two differences between the two processes. The first one has to do with the fact that executing `fork(x)` has as a side effect that a value is assigned to x. In the mother process the assignment `x:=1` is performed, in the daughter process the value 0 is assigned to x. The other difference has to do with the input and output channels of the original process. On execution of the `fork` statement a new intermediate channel is constructed which behaves like a unix pipe. The mother process remains connected to the original input channel, but from now on writes on the new intermediate channel. The daughter will write on the original output channel, but reads from the intermediate channel. The effect of a fork is depicted in figure 4.

An example is the program

```
read(x); write(x);
fork(y);
if y=1
   then while true do
       begin read(x); if even(x) then write(x) end
   else while true do
       begin read(x); if (x mod 3) = 0 then write(x) end.
```

The original process passes one value from input to output unaltered, and then splits into two filters: the mother filters out all odd numbers, passing only the even input numbers to the daughter. The daughter filters out all numbers which are not a multiple of 3. The effect is a filter that passes its first input number unaltered, and then passes only those input value that are multiples of 6.

Another example is the parallel sieve of Eratosthenes:

```
while true do
begin read(x); write(x);
    fork(y);
    if y=1 then
        while true do
        begin read(z);
            if (z mod x) <> 0 then write(z)
        end
end.
```

If on the input channel for the original process the stream $2, 3, 4, 5, \ldots$ is inserted, then execution of this program will result in an expanding array of processes which in cooperation yield an output stream consisting of all prime numbers. The original process can be called an "expander", it reads a number x and expands into a filter process (the mother) which blocks all multiples of x, and a new expander process (the daughter) which behaves exactly like the mother. In figure 5 I show how this networks evolves.

I now turn to the semantics of this language, starting again with continuation semantics. Evaluation of a statement **S** will in the end yield an answer, and first of all I will investigate what kind of answers are feasible here. In all continuation semantics discussed before these answers were states with possibly some extra information added, like nontermination ($\perp$), the error result (1.2.6) or failure/success (1.3.1.2).

In this case the final state is not so interesting mainly because this will not provide enough information in case the computation does not terminate. The information of interest is the output from the computation, not whether the computation terminated or not. This leads us to the following definition of the domain of answers: $A = V^{\infty}$ (cf. section 1.2.4 for the notation).

The output generated by a statement **S** depends of course on the input for **S**, and also on the initial values the variables have, i.e. on the initial state. So the meaning function will have functionality

$$\mathbf{N}\colon \mathbf{Stat} \to \mathbf{Cont} \to \Sigma \to V^{\infty} \to V^{\infty},$$

where **Cont** is the domain of the continuations.

A continuation θ shall be a function that yields a final answer, i.e. a value in V^{∞} which models the result of the computation to be performed once **S** has been evaluated. This means that the effects brought about by executing **S** must be arguments of this continuation. Execution of **S** changes the initial situation in two ways: first of all the initial state σ is transformed into a new state σ', but also the contents of the input channel is affected if during evaluation of **S** input statements have been executed. In that case the initial contents τ of the input channel is transformed (truncated)

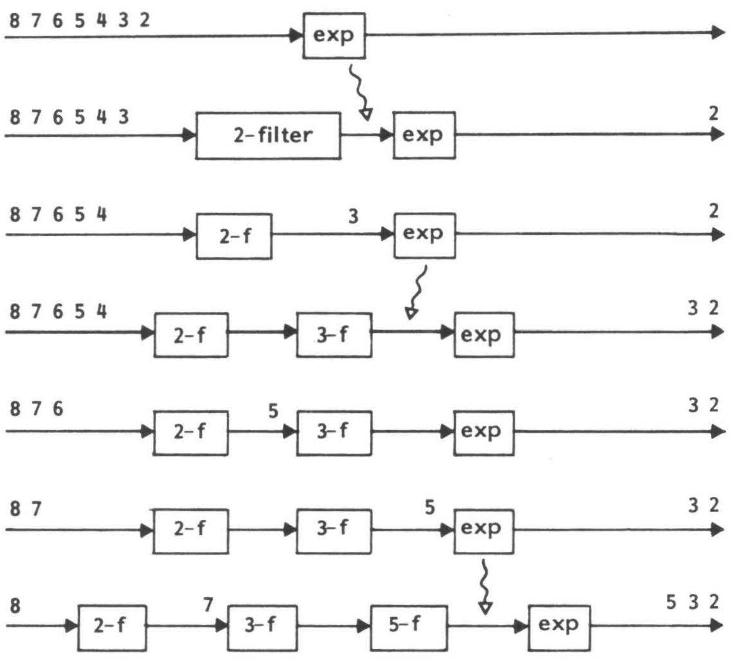

Figure 5. The parallel sieve of Eratosthenes.

as well, yielding a new contents τ'. This leads us to the following functionality of continuations

$$\mathbf{Cont} \;=\; \Sigma \to V^{\infty} \to V^{\infty}.$$

Now that the domains have been determined I can give the semantic clauses:

1. $\mathbf{N}[\![\mathtt{x}:=\mathtt{t}]\!]\theta\sigma\tau \;=\; \theta(\sigma[\mathbf{M}[\![\mathtt{t}]\!]\sigma/\mathtt{x}])\tau$

This is straightforward, evaluation of an assignment transforms only the state and not the contents of the input channel. The next two clauses are standard.

2. $\mathbf{N}[\![\mathtt{S}_1;\mathtt{S}_2]\!]\theta\sigma\tau \;= \mathbf{N}[\![\mathtt{S}_1]\!]\,\{\mathbf{N}[\![\mathtt{S}_2]\!]\theta\}\,\sigma\tau$
3. $\mathbf{N}[\![\,\mathtt{if}\ \mathbf{b}\ \mathbf{then}\ \mathtt{S}_1\ \mathtt{else}\ \mathtt{S}_2\,]\!]\theta\sigma\tau \;=$
   $(\mathbf{M}[\![\mathbf{b}]\!]\sigma \;=\; tt) \;\to\; \mathbf{N}[\![\mathtt{S}_1]\!]\theta\sigma\tau,\; \mathbf{N}[\![\mathtt{S}_2]\!]\theta\sigma\tau$
4. $\mathbf{N}[\![\,\mathtt{while}\ \mathbf{b}\ \mathbf{do}\ \mathtt{S}\,]\!]\theta\sigma\tau \;=$
   $(\mathbf{M}[\![\mathbf{b}]\!]\sigma \;=\; tt) \;\to\; \mathbf{N}[\![\mathtt{S}]\!]\,\{\mathbf{N}[\![\,\mathtt{while}\ \mathbf{b}\ \mathbf{do}\ \mathtt{S}\,]\!]\theta\}\,\sigma\tau,\; \theta\sigma\tau$

There is a familiar circularity in this definition, and here again the desired solution can be obtained through iteration. For reasons similar to the ones in 1.2.4 this iteration should have θ_0 as a starting value, where $\theta_0\sigma\tau$ is defined through $\theta_0\sigma\tau = \langle\,\rangle$ for all σ and τ.

5. $\mathbf{N}[\![\,\mathtt{write(t)}\,]\!]\theta\sigma\tau \;=\; \langle \mathbf{M}[\![\mathtt{t}]\!]\sigma\rangle \mathbin{\hat{}} \theta\sigma\tau$

Outputting the value of **t** in state σ with input τ, followed by a computation defined through θ yields an output sequence which has the value of **t** as its first element, while its tail consists of the values written by θ.

6. $\mathbf{N}[\![\,\mathtt{read(x)}\,]\!]\theta\sigma\tau \;= (\tau = \langle\,\rangle) \;\rightarrow \langle\,\rangle,\; \theta(\sigma[\mathit{first}\;\tau\,/\,\mathrm{x}])(\mathit{rest}\;\tau)$

Input of a value changes the sequence on the input channel, the first element is removed. It also changes the state because this first element is assigned to a variable. However, it is possible that the input channel is empty. In that case the computation blocks and the overall result of the computation will therefore be an empty output stream.

7. $\mathbf{N}[\![\,\mathtt{fork(x)}\,]\!]\theta\sigma\tau \;=\; \big(\theta(\sigma[0/\mathrm{x}])\big)\big(\theta(\sigma[1/\mathrm{x}])\tau\big)$

This is the most interesting clause. Execution of a **fork** statement is a complicated operation, but apparently this can be described quite succinctly in continuation semantics. I summarize: the effect of a **fork** is that two copies of the original process will be around, the mother for which x = 1 holds, and the daughter which has x set to 0. The mother will therefore proceed with the computation in the transformed state $\sigma[1/\mathrm{x}]$, reading input values from the original input stream. The output sequence generated by the mother process is therefore given by $\theta(\sigma[1/\mathrm{x}])\tau$. The computation performed by the daughter process will start at the statement following the **fork** statement, this computation is therefore determined by θ. The initial state for this computation will be the state of the original process with x set to 0. The output generated by the combination of these two processes is the output generated by the daughter, which equals $\theta(\sigma[0/\mathrm{x}])\tau'$, where τ' is the contents of the input channel of the daughter process. This however is equal to the output sequence generated by the mother process, so $\tau' \;=\; \theta(\sigma[1/\mathrm{x}])\tau$.

Having constructed a continuation semantics, the next thing to investigate is whether it is possible to devise a direct semantics for this language. I first consider the semantics of input and output only, so for the moment the **fork** statement is removed from the language.

There are several obstacles to cope with. The first one is that a computation can be blocked because a process tries to read from an empty input channel. I choose the standard solution for this similar to what I did in section 1.2.6 with error exits: I introduce a special state BLOCKED and I must then take care that the meaning $\mathbf{M}[\![\mathtt{S}]\!]$ of all statements **S** will be BLOCKED-strict.

But a further extension of the intermediate states is needed. Evaluation of $\mathtt{S}_1$ in $\mathtt{S}_1;\mathtt{S}_2$ does not only change the proper state (the value of the variables), it has its effects on the input and output channels as well. This means that the extended state to be passed to $\mathtt{S}_2$ must also contain the new contents of the input and the output channel. Combining the above considerations I arrive at the follow definition of the class **Exst** of extended states:

$$\mathbf{Exst} \;=\; (\Sigma_\perp \;\cup\; \{BLOCKED\}) \;\times\; V^\infty \;\times\; V^\infty.$$

Let us try to define the function **M** giving the direct semantics. First of all I have to take care that all meanings are $\perp$- and BLOCKED-strict. Of course it is possible to include this in all semantic clauses, but it is more convenient to state it in advance:

For all $\mathtt{S}$ I have
$\mathbf{M}[\![\mathtt{S}]\!]\langle\perp, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle \;=\; \langle\perp, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle$ and
$\mathbf{M}[\![\mathtt{S}]\!]\langle\mathrm{BLOCKED}, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle \;=\; \langle\mathrm{BLOCKED}, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle.$

Now I can write down the clauses for all the other possible forms of the extended states:

$\mathbf{M}[\![\mathtt{x:=t}]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle \;=\; \langle\sigma[\mathbf{M}[\![\mathtt{t}]\!]\sigma/\mathtt{x}],\ \tau_{\mathrm{in}},\ \tau_{\mathrm{out}}\rangle$
$\mathbf{M}[\![\mathtt{S}_1;\mathtt{S}_2]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle \;=\; \mathbf{M}[\![\mathtt{S}_2]\!](\mathbf{M}[\![\mathtt{S}_1]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle)$
$\mathbf{M}[\![\ \mathtt{if\ b\ then\ S}_1\ \mathtt{else\ S}_2\ ]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle \;=$
$\quad(\mathbf{M}[\![\mathtt{b}]\!]\sigma \;=\; \mathrm{tt}) \;\rightarrow\; \mathbf{M}[\![\mathtt{S}_1]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle,\ \mathbf{M}[\![\mathtt{S}_2]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle$
$\mathbf{M}[\![\ \mathtt{while\ b\ do\ S}\ ]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle \;=$
$\quad(\mathbf{M}[\![\mathtt{b}]\!]\sigma \;=\; \mathrm{tt}) \;\rightarrow\; \mathbf{M}[\![\ \mathtt{while\ b\ do\ S}\ ]\!](\mathbf{M}[\![\mathtt{S}]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle),\ \langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle$
$\mathbf{M}[\![\ \mathtt{read(x)}\ ]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle \;=$
$\quad(\tau_{\mathrm{in}} = \langle\,\rangle) \;\rightarrow\; \langle\mathrm{BLOCKED}, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle,\ \langle\sigma[\mathit{first}\ \tau_{\mathrm{in}}\ /\ \mathtt{x}], \mathit{rest}\ \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle$
$\mathbf{M}[\![\ \mathtt{write(t)}\ ]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle \;=\; \langle\sigma,\ \tau_{\mathrm{in}},\ \tau_{\mathrm{out}}\rangle^\wedge\langle\mathbf{M}[\![\mathtt{t}]\!]\sigma\rangle\ \rangle$

These clauses look only a little bit more complicated than the corresponding clauses for the continuation semantics, but there is more going on behind the scenes than can be seen at first glance. Some of the difficulties show up if one considers the clause on the `while` statement. This clause again defines a fixed point, a solution of this equation, and this solution will as always be equal to the limit of a chain of approximations. Some of the complexities now come to the fore: one should be careful in choosing the first approximation ϕ_0.

$\phi_0\langle\mathrm{BLOCKED}, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle \;=\; \langle\mathrm{BLOCKED}, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle$
$\phi_0\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle \;=\; \langle\perp, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle$

In more technical terms, it is not obvious how to find a suitable ordering on **Exst** to make the operator continuous of which $\mathbf{M}[\![\mathtt{while\ b\ do\ S}]\!]$ must be the fixed point.

Next a few words on the direct semantics of the `fork` construct. We have to find a sensible definition of $\mathbf{M}[\![\ \mathtt{fork(x)}\ ]\!]\langle\sigma, \tau_{\mathrm{in}}, \tau_{\mathrm{out}}\rangle$, but it is not very clear how to obtain this in an elegant way. Once more the state will have to be extended, because the fact that the process has been split in two must be passed to the next statement.

A tentative solution is to pass not one extended state but two instead, one describing the mother process and the other describing the daughter process. The state for the mother process would be something like $\langle\sigma[1/\mathrm{x}], \tau_{\mathrm{in}}, \langle\,\rangle\,\rangle$, because this process keeps the original input channel but obtains a new output channel which is of course empty immediately after the `fork`. The extended state for the daughter will be something like $\langle\sigma[0/\mathrm{x}], X, \tau_{\mathrm{out}}\rangle$, because the daughter inherits the output channel.

Now consider the sequence X, which should consist of the values the mother process will produce. The problem is that at the moment of expansion this sequence is not known and therefore cannot be passed to the next statement. A possible solution might be to use abstraction and to pass, instead of this extended state, a function taking an input stream X as an argument and yielding an extended state. Although it might be possible to obtain a direct semantics working along these lines, the result will undoubtedly be much less perspicuous than the continuation semantics given before.

All this can be contrasted with the relative simplicity of the clause for the `fork` construct using continuation semantics. This simplicity stems from the fact that in continuation semantics one has the denotation of the future of the computation at one's disposal, so there is sufficient information available to determine the output streams generated by the mother and the daughter process.

1.3.1.4. Some final remarks. The essential difference between direct and continuation semantics lies in the respective definitions of the meaning of composition of statements:

$$\mathbf{M}[\![\, \mathrm{S}_1; \mathrm{S}_2 \,]\!]\sigma \;=\; \mathbf{M}[\![\mathrm{S}_2]\!](\mathbf{M}[\![\mathrm{S}_1]\!]\sigma)$$

versus

$$\mathbf{N}[\![\, \mathrm{S}_1; \mathrm{S}_2 \,]\!]\theta\sigma \;=\; \mathbf{N}[\![\mathrm{S}_1]\!]\{\mathbf{N}[\![\mathrm{S}_2]\!]\theta\}\sigma$$

Direct semantics forces one to apply the meaning $\mathbf{M}[\![\mathrm{S}_2]\!]$ to some intermediate result $(\mathbf{M}[\![\mathrm{S}_1]\!]\sigma)$ even if this is not feasible, for instance because S_2 is a **goto** statement. In continuation semantics there is more flexibility because the meaning of S_2 is passed to $\mathbf{N}[\![\mathrm{S}_1]\!]$ as a parameter, and now there are several options on how to deal with this parameter.

Evaluation of S_1 will yield some intermediate results and if S_1 is such that its execution will not disrupt the normal flow of control, then the continuation will be applied to these intermediate values. This case is similar to the way direct semantics handles composition. The alternative is that $\mathbf{N}[\![\mathrm{S}_1]\!]$ ignores its continuation, which will happen if S_1 is a **goto** statement. This idea can be refined, because there is also the option to not use the continuation for the moment, but to keep it for later use instead. Breaks from loops can be modelled in this way, and so can coroutines.

The possible effects of execution of a statement are threefold. Firstly, intermediate result will be generated that will be used later in the computation. For instance, some variables will obtain a new value, or the sequence on the input channel might be truncated (section 1.3.1.3). Another example of this can be found in the backtrack language (1.3.1.2): new elements might be added to the set of alternatives. Secondly, evaluation of a statement might produce part of the answer, the final outcome of the whole computation. An example of this is the output stream in the previous section. Finally, the standard sequencing of statement evaluation can be altered through jumps, error exits and the like.

Because of the fact that in direct semantics the operator ; for composition of statements is modelled through composition of functions, all the above effects have to be coded into intermediate extended states. These extended states will therefore be tuples existing of at least three components: the intermediate results (the new values of the variables and possibly other entities as well, like the current sequence on the input channel), the fragment of the final answer produced up till now, and optional sequencing information indicating whether execution should proceed in the standard way with the textually next statement. From the results of the preceding sections one can conclude that these extended states tend to grow into complex objects. The consequence is that in the semantic clauses many subcases have to be distinguished, and another effect is that the domains have to be set up carefully in order still to be able to apply Scott's theory.

In continuation semantics each of these different sort of effects produced by execution of a statement is modelled separately. First the intermediate results. These are handled by making the continuations functions that accept an argument for each kind of intermediate result that can be produced by evaluation of a statement. For example, in the language from section 1.3.1.2 execution of a statement produces an intermediate state in Σ and also a new set of alternatives, therefore the continuations will take such a state and a denotation of these alternatives as arguments: $\mathbf{Succ} = \mathbf{Alt} \rightarrow \Sigma \rightarrow A$.

Next the pieces of the final answer. In continuation semantics the meaning $\mathbf{N}[\![\mathtt{S}]\!]$ of S is a function which in the end will yield an answer, the final result of the whole computation. Evaluation of S might produce a fragment of this answer. We saw that in direct semantics this fragment must be passed to the next statement. In continuation semantics however a denotation of all statements to be executed after evaluation of S is available. It is therefore possible to directly combine the piece of the answer generated by S with the rest to be generated by the continuation. A particularly nice illustration of this is the clause on the `fork` construct in section 1.3.1.3.

Finally, because the continuation is an argument of the meaning $\mathbf{N}[\![\mathtt{S}]\!]$ of a statement, a deviation in the flow of control from the standard sequencing can be handled straightforwardly. One of the consequences of this is the fact that in continuation semantics there is no need to add $\perp$ to the set of states, while in direct semantics this must be done.

The disadvantage of continuation semantics is that the flow of control is captured in a more indirect way than direct semantics does. Therefore it takes some thinking before one has digested the underlying ideas and the "back-to-front" manner in which the meaning functions are derived from program texts. For those simple languages where the apparatus of direct semantics is powerful enough, there is no reason to introduce this complexity from continuation semantics.

However we saw that for each of the concepts discussed in the preceding sections, its introduction in the language tends to add substantial complexity to the clauses of direct semantics. And it was also clear that the additional options offered by continuations led to semantic clauses that remained compact and elegant. It therefore pays off already very soon to switch from direct to continuation semantics.

1.3.2. CONTINUATIONS, IMPLEMENTATIONS AND PROOF SYSTEMS

One of the intentions of the founders of denotational semantics was that it should be used as a standard against which implementations can be verified [ScS71]. A straightforward way to use denotational semantics in implementing a language is to transform it into an interpreter. It is possible to use the clauses of the semantics as input for an "interpreter generator", for instance Mosses' SIS system [Mos75, Mos79].

Another approach might be to construct an interpreter by hand which accepts a program and input for this program, and then "executes" the program according to the clauses of the denotational semantics for the language in which the program is written, more or less like I evaluated in section 1.2.4 a little program according to the Cook semantics for WHILE. A problem here is that denotational semantics uses abstract, and often infinite entities, like functions.

A solution might be to use representations instead of the abstract objects. For instance, the continuation semantics of section 1.2.6 acts on states σ, and continuations θ. However states might be represented by finite sequences consisting of variable-value pairs, and good representations of the continuations which are built up by the semantics will be sequences of statements. The semantic clauses of 1.2.6 must of course be adapted because of this change, but this can be done straightforwardly. The resulting semantics will be close to the operational semantics presented in section 1.2.5 and this semantics lends itself readily to be interpreted. So apparently, if it is possible to find suitable representations of the abstract objects manipulated by a denotational semantics then this semantics can be transformed into an interpreter.

In the same way I have constructed an interpreter (operational semantics) for the SNOBOL4 subset in chapter 3, section 3 from the denotational semantics which is introduced in section 4 of the same chapter. Finding the right representations for the continuations occurring in this denotational semantics demanded more thought than it did in the case of the language WHILE above, which is visible in the resulting operational semantics. However if the denotational semantics is well understood, the right representations can be found, because these must be manipulated by the interpreter in the same spirit as the continuations are transformed in the denotational semantics. For a much more intricate language the same method has been used by Stoy [Sto81], the interpreter introduced in this paper can be derived from the continuation semantics presented there in the way I just described.

Finding good representations of the abstract objects occurring in the denotational semantics requires some ingenuity. Simple objects like states which are functions from simple sets to simple sets and which are at any time defined on a finite subset of their domain only, can be readily represented by finite sequences as indicated above. For higher order functions this is more intricate. This is one of the arguments against continuations that are brought forward by workers from the VDM group [Bjø80]: "Continuations define a semantics a bit more implicitly, and one needs 'realize' continuations through some more mechanical device for it to be machine-processable".

Of course interpreters are not efficient and this is even more so for interpreters obtained as described above. Efficient implementations must be realised by compilation. Also in this case an implementation might be justified against the denotational standard. This task has now become a lot more complicated. An implementation is concrete: the objects involved are representations (bit patterns), procedures are represented by sequences of machine instructions, there is a lot of bookkeeping going on, room for variables and intermediate results is allocated and freed dynamically on the runtime stack. On the other hand, a denotational definition of a language is abstract: the objects that are manipulated are mathematical entities, functions are defined not by describing how they are evaluated but through fixed points, solutions of circular equations, and there is the deliberate choice to hide as much bookkeeping as possible.

In [MiS76] a proof is given of correctness of a compiler for SAL, a language with the complexity of Algol68. This proof is long, impressive and also intimidating (2 volumes, 858 pages). The complexity is caused by the fact that the source language is complex, and also that the semantics of SAL is not related to the compiler and the target machine. From this work one might conclude that it is not a promising idea to try to validate an implementation a posteriori.

More modern work concentrates on how an implementor can be guided in his work by the denotational definition of the language he is working on. Tools and methods are developed which can be used in the transformation of a denotational definition into a compiler in a correctness preserving way [Mos80, Wan82, Mos83, Sch85, JSS85]. It is even possible to take one further step and develop "compiler generators" systems that accept a denotational definition and generate a compiler from this [JoS80, ChJ83, Set83].

The rest of this section will be devoted to a few remarks on how denotational semantics can be used to justify Hoare like proof systems.

A Hoare triple is a syntactic object of the form $\{p\}S\{q\}$, where S is a statement and p and q are formulae from first order predicate logic (boolean expressions with quantifiers), which describe properties of the values of variables that can occur in S. Such a triple can be used to describe the effect execution of S will have on the values of its variables, the meaning of such a triple (validity) is: if p holds for the values of the variables before execution of S, and this execution terminates normally then q must be true of the resulting values of the variables (partial correctness).

This notation was introduced in [Hoa69], together with a proof system for several programming constructs which can be used to derive a formal proof for such triples. In the years to follow rules for other constructs were suggested, and also papers appeared that tried to justify such proof systems. A survey paper is [Ap81b], and [Bak80] contains a thorough treatment of these issues.

The majority of the programming constructs involved were statements of the single-entry single-exit type. This is not very surprising if one considers the definition of validity of Hoare triples given above. The natural consequence is that the justifications of these systems were based on models constructed with direct semantics.

However, already in 1972 Clint and Hoare presented the paper [ClH72] introducing a proof rule for the **goto** statement. I study this and a related system in chapter 4, and here I will present an introduction to these results. The proof rule in [ClH72] is:

$$\frac{\{\mathtt{q}\}\mathbf{goto}\,\mathtt{L}\{\mathtt{false}\} \vdash \{\mathtt{p}\}\mathtt{S}\{\mathtt{q}\}, \quad \{\mathtt{q}\}\mathbf{goto}\,\mathtt{L}\{\mathtt{false}\} \vdash \{\mathtt{q}\}\mathtt{S}'\{\mathtt{r}\}}{\vdash \{\mathtt{p}\}\mathtt{S};\ \mathtt{L}{:}\,\mathtt{S}'\{\mathtt{r}\}}$$

where it is assumed that all **goto** statements occurring in S and S′ specify a jump to label L, and also that no other labels are defined in S and S′. The above rule means that if you can prove both $\{p\}S\{q\}$ and $\{q\}S'\{r\}$ using the assumption $\{q\}$**goto** L$\{\mathtt{false}\}$, then you have in fact a proof of $\{p\}S;\ L{:}\,S'\{r\}$.

Now this rule is not easily understood: what is the meaning of a Hoare triple now that there are jumps in the language, is there a model in which validity can be formulated of rules specifying proofs from assumptions? In chapter 4 the above rule is justified using a definition of validity based on continuation semantics. In that chapter I also present another proof system that is operationally equivalent with the one above in the sense that proofs conducted in one system can be mechanically carried over to the other system.

This other system is similar to the system in [ArA79]. The underlying idea is to use the single-entry multi-exit model of the meaning of statements and to introduce an extended version of Hoare formulae which corresponds to this model. These formulae now have the form $\{p\}S\{q\}\{L{:}\,r\}$. The meaning of such a formula is: if p holds for the values of the variables before execution of S and S terminates normally then q must hold afterwards, and if S terminates through a jump to L then after this jump r must hold. It will be clear that this validity definition can be justified in a straightforward manner using the semantic function **A** from section 1.3.1.1. Based on such a definition of validity I justify a variant of this system in section 7 of chapter 4.

Notice that the formula $\{p\}$**goto** L$\{\mathtt{false}\}\{L{:}\,p\}$ is trivially true under this definition of validity. The central proof rule given in [ArA79] is

$$\frac{\{\mathtt{p}\}\mathtt{S}\{\mathtt{q}\}\{\mathtt{L}{:}\,\mathtt{q}\},\quad \{\mathtt{q}\}\mathtt{S}'\{\mathtt{r}\}\{\mathtt{L}{:}\,\mathtt{q}\}}{\{\mathtt{p}\}\mathtt{S};\ \mathtt{L}{:}\,\mathtt{S}'\{\mathtt{r}\}}$$

again assuming that the only label name occurring in **goto**'s in S and S′ is L, and that no other labels are defined in S or S′. The correctness of this rule can be seen by considering an execution of S; L: S′ starting in a state for which p holds. Then we have to show that after termination r holds. The crucial observation is that the assumptions of the proof rule guarantee that every time that during evaluation of S; L: S′ control arrives at L that then q must hold. This can be shown by induction on the number of times control arrives at L.

In order to get some insight into how this system is used I present a program describing a non standard way to set a variable to 0:

$$\mathtt{S} \equiv \mathtt{if}\ x<0\ \mathtt{then}\ x:=-x;\ L\colon \mathtt{if}\ x<>0\ \mathtt{then}\ (x:=x-1;\ \mathtt{goto}\, L)$$

I have to prove $\{\mathtt{true}\}\,\mathtt{S}\,\{x=0\}$, and I will use the label invariant that at L always $x\geq 0$ holds. First of all I have

$\{x<0\}\ x:=-x\ \{x\geq 0\}$

from which

$\{\mathtt{true}\}\ \mathtt{if}\ x<0\ \mathtt{then}\ x:=-x\ \{x\geq 0\}$

can be deduced, and thus a fortiori

$\{\mathtt{true}\}\ \mathtt{if}\ x<0\ \mathtt{then}\ x:=-x\ \{x\geq 0\}\{L\colon x\geq 0\}$

Furthermore I have

$\{x>0\}\ x:=x-1\ \{x\geq 0\}$

if x is an integer. Also

$\{x\geq 0\}\ \mathtt{goto}\, L\ \{\mathtt{false}\}\{L\colon x\geq 0\}$

which combines to

$\{x>0\}\ x:=x-1;\ \mathtt{goto}\, L\ \{\mathtt{false}\}\{L\colon x\geq 0\}$

This then justifies

$\{x\geq 0\}\ \mathtt{if}\ x<>0\ \mathtt{then}\ (x:=x-1;\ \mathtt{goto}\, L)\ \{x=0\}\{L\colon x\geq 0\}$

Now the proof rule from [ArA79] can be applied which yields the desired result.

This system and the one by Clint and Hoare are closely connected: essentially the same proof can be given using the latter system. This observation is used in chapter 4 to justify the Clint-Hoare system. The main differences between the systems is that the Clint-Hoare system implicitly keeps track of the label invariants, while the above system does so explicitly.

The relation between the systems in [ClH72] and [ArA79] bears some resemblance to the relation between continuation and direct semantics. The introduction of **goto**'s in the latter proof system is accommodated by an extension of the correctness formulae which is similar to the way the states are extended in VDM semantics. Therefore the [ArA79] style has the same disadvantages as direct semantics: if another concept has to be adopted in the system that also does not fit in the single-entry single-exit scheme, then again the syntax of the formulae has to be extended. Consider for instance coroutines. The [ArA79] formalism has to be extended because the targets of the jumps are anonymous now, whereas in [Cli83] a rule is suggested which is expressed in the same framework as [ClH72].

Therefore it seems to be worthwile to try to give a straightforward definition of validity of the formulae in the Clint-Hoare system, which can then be used to justify this system in a direct way. In section 8 of chapter 4 such a validity definition is proposed, and used to prove soundness of the system (Theorem 8.5).

This validity definition also refutes an objection against the Clint-Hoare system raised in [Don82], which I will now discuss. If one wants to validate or invalidate a proof system, one needs to have a definition of validity of the formulae occurring in the system. An obstacle in this respect are the formulae with assumptions:

$$\{p\}\mathbf{goto}\,L\{\mathtt{false}\} \;\vdash\; \{q\}S\{r\},$$

because such a formula expresses a property of the proof of $\{q\}S\{r\}$, it states that a formal proof is possible if one can use the assumption $\{p\}\mathbf{goto}\,L\{\mathtt{false}\}$. The $\vdash$-symbol denotes a property that has no counterpart in a model. To circumvent this I do not allow such formulae in the system, but replace these by formulae of the form

$$\{p\}\mathbf{goto}\,L\{\mathtt{false}\} \;\Rightarrow\; \{q\}S\{r\},$$

and I rephrase all proof rules and axioms of the original system in these terms. This yields a system that is operationally equivalent, but for which I can provide validity definitions. For instance a natural definition is that the above formula is valid if truth of $\{p\}\mathbf{goto}\,L\{\mathtt{false}\}$ implies truth of $\{q\}S\{r\}$.

Now both in [ArA79] and in [Don82] the observation is made that according to the original validity definition of Hoare triples as given above, $\{p\}\mathbf{goto}\,L\{\mathtt{false}\}$ is valid for all p. This is so, because there is no normal termination, the formula stipulates that **false** must be true only if control reaches the "right hand end" of **goto** L which it never does.

But this implies that $\{p\}\mathbf{goto}\,L\{\mathtt{false}\} \Rightarrow \{q\}S\{r\}$ is valid whenever $\{q\}S\{r\}$ is valid. In other words, the assumptions allowed in the Clint-Hoare system are irrelevant as far as validity of the formulae is concerned. On the other hand, the assumptions play an essential role in the proofs: the only formulae on a **goto** statement that can be derived in the system are

$\{p\}\mathbf{goto}\,L\{\mathtt{false}\} \Rightarrow \{p\}\mathbf{goto}\,L\{\mathtt{false}\}$

and weakened versions thereof.

In fact, this is the mechanism that makes the Clint-Hoare system "theorem sound" [Don82]: all formulae $\{p\}S\{q\}$ without assumptions that can be derived in the system are valid.

However, there exist valid formulae containing assumptions:

$\{\mathtt{false}\}\mathbf{goto}\,L\{\mathtt{false}\} \Rightarrow \{\mathtt{true}\}\mathbf{goto}\,L\{\mathtt{false}\}$

$\{\mathtt{false}\}\mathbf{goto}\,L\{\mathtt{false}\} \Rightarrow \{\mathtt{false}\}\mathtt{skip}\{\mathtt{false}\}$

which can be combined using the Clint-Hoare rule to the invalid formula

$\{\mathtt{true}\}\mathbf{goto}\,L;\ L{:}\,\mathbf{skip}\{\mathtt{false}\}$

This means that the system is not "inferentially sound" and in [Don82] it is explained eloquently why this is a bad thing.

There is however a flaw in this reasoning and that is the starting point, the definition of validity of Hoare triples. The assumption in $\{p\}\mathbf{goto}\,L\{\mathtt{false}\} \Rightarrow \{q\}S\{r\}$ must not be interpreted according to the classical definition which makes it meaningless. Such an assumption defines a property of the environment in which S is executed, i.e. it states a property of the meaning of the label L. Validity of {p}goto L{false} should be formulated with respect to the denotation of L, it should put a restriction on the continuations which are acceptable denotations of L.

In this way validity of the formula {p}goto L{false} $\Rightarrow$ {q}S{r} can be formulated as "if the environment in which S is executed is such that {p}goto L{false} holds, then {q}S{r} will be true in that environment". This is an approach similar to the one in the justification of the proof rules for recursive procedures [Ap81b], [Bak80].

The definition of validity presented in section 8 of chapter 4 is refined enough to make the Clint-Hoare system inferentially sound. In particular, validity is defined such that the formula

{false}goto L{false} $\Rightarrow$ {true}goto L{false}

from the counterexample above is no longer valid.

Although my validity definition is more complicated than the standard one (due to the fact that it uses continuations), I expect that in the end this will pay off, because essentially the same definition might be used to justify proof rules for other statements that specify non standard sequencing, which is something that cannot be expected for the systems that can be justified using direct semantics. It might very well be that the same phenomenon can be observed in relation to proof rules as I discussed in 1.3.1 for semantics in general.

REFERENCES

[Ap81a] K.R. Apt, "Recursive assertions and parallel programs", *Acta Informatica* **15** (1981), 219–232

[Ap81b] K.R. Apt, "Ten years of Hoare's logic: a survey— part I", *ACM Trans.. Prog. Lang. and Syst.* **3** (1981), 431–483

[ArA79] M.A. Arbib and S. Alagić , "Proof rules for gotos", *Acta Informatica* **11** (1979), 139–148

[Bak80] J.W. de Bakker, "Mathematical theory of program correctness", Prentice Hall Int., London (1980)

[BBH74] H. Bekić , D. Bjørner, W. Henhapl, C.B. Jones and P. Lucas, "Formal definition of a PL/I subset", IBM Vienna Lab. Techn. Report TR25.139 (1974)

[BjJ78] D. Bjørner, C.B.Jones (eds.), "The Vienna Development Method: the Meta-Language", LNCS 61, Berlin Heidelberg New York, Springer (1978)

[BjO80] D. Bjørner, O.N. OEST (eds.),"Towards a formal description of Ada", LNCS 98, Berlin Heidelberg New York, Springer (1980)

[Bjø80] D. Bjørner, "Experiments in block-structured goto language modelling: exits versus continuations", in: Winter school on formal specification methods (D. Bjørner, ed.), LNCS 86, Berlin Heidelberg New York, Springer (1980)

[ClH72] M. Clint, C.A.R. Hoare, "Program proving: jumps and functions", *Acta Informatica* **1** (1972), 214–224

[Cli73] M. Clint, "Program proving: coroutines", *Acta Informatica* **2** (1973), 50–63

[Coh79] J. Cohen, "Non-deterministic algorithms", *Comp. Surveys* **11** (1979), 79–94

[Coo78] S.A. Cook, "Soundness and completeness of an axiom system for program verification", *SIAM J. Comp.* **7** (1978), 70–90

[Don82] M.J. O'Donnell, "A critique of the foundations of Hoare style programming logics", *CACM* **25** (1982), 927–935

[Gim73] J.F. Gimpel, "A theory of discrete patterns and their implementation in SNOBOL4", *CACM* **16** (1973), 91–100

[Gim75] J.F. Gimpel, "Nonlinear pattern theory", *Acta Informatica* **4** (1975), 213–229

[Gor73] M.J.C. Gordon, "Models of pure LISP (a worked out example in semantics)", Experimental programming reports 31, Dept. of machine intelligence, Univ. of Edingburgh (1973)

[HaB81] P.L. Haff and D.Bjørner (eds.), "The formal definition of CHILL", C.C.I.T.T. Recommendation Z200 supplement, ITU Geneva (1981)

[HeJ78] W. Henhapl and C.B. Jones, "A formal definition of Algol60 as described in the 1975 modified report", in: [BjJ78] (1978)

[HeP79] M.C.B. Hennessy and G.D. Plotkin, "Full abstraction for a simple programming language", in: Proc. 8th symp. on mathematical foundations of computer science, LNCS 74, Berlin Springer (1979), 108–120

[Hoa69] C.A.R. Hoare, "An axiomatic basis for computer programming", *CACM* **12** (1969), 576–580,583

[Jan83] T.M.V. Janssen, "Foundations and applications of Montague grammar", Ph.D. Thesis, Univ. of Amsterdam, Amsterdam (1983)

[Jon78] C.B. Jones, "Denotational semantics of goto: an exit formulation and its relation to continuations", in: [BjJ78] (1978)

[JoS80] N.D. Jones and D.A. Schmidt, "Compiler generation from denotational semantics", in: Semantics directed compiler generation (N.D. Jones, ed.), LNCS 94, Berlin Heidelberg New York, Springer (1980), 70–93

[JSS85] N.D. Jones, P. Sestoft and H. Søndergaard, "An experiment in partial evaluation: the generation of a compiler generator", in: Rewriting techniques and applications (J. Jouannaud, ed.), LNCS 202, Berlin Heidelberg New York, Springer (1985), 124–140

[Kah74] G. Kahn, "The semantics of a simple language for parallel programming", in: J.L. Rosenfeld (ed.), IFIP74, Amsterdam, North-Holland Publ. Comp. (1974), 471–475

[KaM77] G. Kahn and D.B. MacQueen, "Coroutines and networks of parallel processes", in: B. Gilchrist (ed.), IFIP77, Amsterdam, North-Holland Publ. Comp. (1977), 993–998

[Lan64] P.J. Landin, "The mechanical evaluation of expressions", *Comp. Journal* **6** (1964), 308–320

[Maz70] A. Mazurkiewicz, "Proving algorithms by tail functions", *Information and Control* **18** (1970), 220–226

[McC63] J. McCarthy, "Towards a mathematical science of computation", in: Information

Processing 1962, Proc. IFIP congress 1962 (M.C. Popplewell, ed.), North-Holland Publ. Co., Amsterdam (1963), 21–28

[Mil74] R.E. Milne, "The formal semantics of computer languages and their implementations", Ph.D. thesis, Univ. of Cambridge, also: Techn. Microfiche TCF-2, Programming Research Group, Univ. of Oxford (1974)

[MiS76] R. Milne and C. Strachey, "A theory of programming language semantics", Chapman & Hall, London and Wiley, New York, 2 Vols. (1976)

[Mor70] F.L. Morris, "The next seven hundred programming language descriptions (typescript)", Computer Centre, University of Essex, Colchester (1970)

[Mos74] P.D. Mosses, "The mathematical semantics of Algol60", Techn. Monograph PRG-12, Programming research group, Univ. of Oxford (1974)

[Mos75] P.D. Mosses, "Mathematical semantics and compiler generation", Ph.D. thesis, Oxford University (1975)

[Mos79] P.D. Mosses, "SIS—semantics implementation system: Reference manual and user guide", DAIMI MD-30, Dept. Comp. Sc., University of Aarhus, Denmark (1979)

[Mos80] P.D. Mosses, "A constructive approach to compiler correctness", in: Automata, languages and programming, 7th coll., (J.W. de Bakker & J. van Leeuwen, eds.), LNCS 85, Springer New York (1980), 449–469

[Mos83] P.D. Mosses, "Abstract semantic algebras!", in: Proc. IFIP working conf. formal description of programming concepts (D. Bjørner, ed.), North-Holland, Amsterdam New York Oxford (1983), 45–71

[Sch85] D.A. Schmidt, "Detecting global variables in denotational specifications",*ACM Trans. Prog. Lang. and Syst.* **7** (1985), 299–310

[Sco70] D. Scott, "Outline of a mathematical theory of computation", Techn. Mon. PRG-2, Oxford University Computing Laboratory, Programming Research Group, (1970)

[ScS71] D. Scott and C. Strachey, "Towards a mathematical semantics for computer languages", in: Proc. Symp. computers and automata, Polytechnic institute of Brooklyn (1971), 19–46; also: Techn. Mon. PRG-6, Oxford Univ. Computing Lab.

[Set83] R. Sethi, "Control aspects of semantics directed compiling", *ACM Trans. Prog. Lang. and Syst.* **7** (1983), 554–595

[Sto77] J.E. Stoy, "Denotational semantics—the Scott-Strachey approach to programming language theory", MIT Press, Cambridge (1977)

[Sto81] J.E. Stoy, "The congruence of two programming language definitions", *Theor. Comp. Science* **13** (1981), 151–174

[StW74] C. Strachey and C. Wadsworth, "Continuations: a mathematical semantics for handling full jumps", Tech. Mon. PRG-11, Oxford University Computing Lab., Programming Research Group (1974)

[Ten73] R.D. Tennent, "Mathematical semantics of SNOBOL4", ACM SIGPLAN/SIGACT Symp. on principles of programming languages, Boston (1973), 95–107

[Ten76] R.D. Tennent, "The denotational semantics of programming languages", *CACM* **19** (1976), 437–453

[Ten77] R.D. Tennent, "A denotational definition of the programming language PASCAL", Techn. Rep. 77-47, Dept. of Computing and Information Science, Queen's Univ., Kingston, Ontario (1977)

[Wan82] M. Wand, "Deriving target code as a representation of continuation semantics", *ACM Trans. Prog. Lang. and Syst.* **4** (1982), 496–517

[Weg72] P. Wegner, "The Vienna definition language", *Comp. Surv.* **4** (1972), 5–63

chapter 2

ON THE EXISTENCE OF COOK SEMANTICS

Reprinted, with kind permission, from SIAM J. COMPUT.,
Vol. 13, February 1984

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

ON THE EXISTENCE OF COOK SEMANTICS*

ARIE DE BRUIN†

Abstract. In [SIAM J. Comput., 7 (1978), pp. 70–90] Cook defines the operational semantics of a programming language in the following way: a function is introduced which takes a program R and a state σ and yields a possibly infinite row of intermediate states as a result. This row is meant to be the trace resulting from executing program R starting in state σ. This function is characterized by a number of equations. However it is not immediately clear whether these equations have a solution. In this paper we show for a simple language, the most sophisticated feature of which is that it has parameterless procedures, that the corresponding equations have a unique solution. The techniques used here can also be applied to other languages described in the same way, for instance to the language in Cook's paper.

Key words. operational semantics, Cook semantics, fixed points, continuation semantics, recursive definitions, denotational semantics

1. The problem. In this paper we investigate a certain way of defining operational semantics of programming languages, which has been introduced by Cook in his paper on soundness and completeness [6]. Cook remarks that this semantics has been derived from one of the operational semantics studied in Lauer's thesis [10], and also in Hoare and Lauer [7], which is a condensed version of the thesis. This style of definition has later on been employed by de Bakker in his book on the theory of program correctness [3].

The technique is as follows: a meaning function **Comp** is described which takes a program and an initial machine state and yields a row of states as a result. This row gives the trace left by evaluating the program starting in the initial state. A terminating computation yields a finite row, and if evaluation does not terminate then the outcome is an infinite row.

We will study Cook semantics using a simple language. Before giving its syntax we introduce some notational conventions.

Rows will be indicated by angular brackets. For instance we have $\langle x_1, \cdots, x_n\rangle$ which denotes a finite row of n elements, and $\langle x_1, x_2, \cdots\rangle$ which denotes an infinite row. The empty row is denoted by $\langle\,\rangle$. Function application associates to the left, that is $fabc$ is an abbreviation of $((f(a))(b))(c)$. Correspondingly, the $\rightarrow$-operator used in forming function domains associates to the right. The above function f should have functionality definition $f: A\rightarrow B\rightarrow C\rightarrow D$, which should read as $f: A\rightarrow(B\rightarrow(C\rightarrow D))$.

We next describe the syntax of the language. We distinguish the following syntactic classes:

$P\in$ **Pvar**	Procedure variables.
$A\in$ **Atst**	Atomic statements. The structure of these statements is not specified further, but think of assignments.
$B\in$ **Bexp**	Boolean expressions. These are also considered to be atomic building blocks.
$R\in$ **Prog**	Programs. These have the form $\langle E\|S\rangle$ and must be closed, i.e. all procedure variables in E and S are declared in E.
$E\in$ **Decl**	Declarations. These have the form $\langle P_1\Leftarrow S_1, \cdots, P_n\Leftarrow S_n\rangle$ where all P_i are different.
$S\in$ **Stat**	Statements. This class is defined by the following *BNF*-like syntax.

$S\in$ **Stat** Statements. This class is defined by the following *BNF*-like syntax.

$S ::= A|P|$ if B then S_1 else $S_2|S_1; S_2$.

* Received by the editors June 10, and in final revised form November 1, 1982.

† Faculty of Economics, Erasmus University, Rotterdam, the Netherlands.

We now turn to the semantics. There are the following semantic classes.

$\sigma \in \Sigma$	States. The internal structure of states is not specified. Notice that Σ is a set, not a cpo. There is for instance no such thing as $\perp$ in Σ.
$\tau \in \Sigma^\infty$	Rows of states. We define $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. Σ^* contains the finite sequences and the empty row and Σ^ω the infinite ones.

We define the following operators on rows of states.

$^\wedge$ Concatenation, defined by the axioms:

$\tau_1{}^\wedge\tau_2 = \tau_1$ for all $\tau_1 \in \Sigma^\omega$

$\langle\,\rangle^\wedge\tau = \tau^\wedge\langle\,\rangle = \tau$

$\langle\sigma_1, \cdots, \sigma_n\rangle^\wedge\langle\sigma'_1, \cdots, \sigma'_k\rangle = \langle\sigma_1, \cdots, \sigma_n, \sigma'_1, \cdots, \sigma'_k\rangle$

$\langle\sigma_1, \cdots, \sigma_n\rangle^\wedge\langle\sigma'_1, \cdots\rangle = \langle\sigma_1, \cdots, \sigma_n, \sigma'_1, \cdots\rangle$

κ Last element extraction, defined by

$\kappa\langle\sigma_1, \cdots, \sigma_n\rangle = \sigma_n$

$\kappa\langle\,\rangle = \kappa\tau = \bar{\sigma}$ for all $\tau \in \Sigma^\omega$, where $\bar{\sigma}$ is an arbitrary (but fixed from now on) element of Σ.

Finally we distinguish the following elementary valuations.

A: **Atst** $\to \Sigma \to \Sigma$, Meaning of atomic statements. Notice that atomic statements always terminate.

B: Bexp $\to \Sigma \to \{tt, ff\}$, Meaning of boolean expressions.

As the internal structure of **Atst** and **Bexp** has not been specified, we cannot do more than postulate the existence of functions **A** and **B** with functionalities as above.

We now have enough tools to formulate the equations which are intended to define a function **Comp**: **Prog** $\to \Sigma \to \Sigma^\infty$.

$\mathbf{Comp}\langle E|A\rangle\sigma = \langle \mathbf{A}A\sigma\rangle$

$\mathbf{Comp}\langle E|P_i\rangle\sigma = \langle\sigma\rangle^\wedge\mathbf{Comp}\langle E|S_i\rangle\sigma$, with $P_i \Leftarrow S_i$ in E.

$$\mathbf{Comp}\langle E|\text{ if } B \text{ then } S_1 \text{ else } S_2\rangle\sigma = \begin{cases}\langle\sigma\rangle^\wedge\mathbf{Comp}\langle E|S_1\rangle\sigma, & \text{if } \mathbf{B}B\sigma = tt\\ \langle\sigma\rangle^\wedge\mathbf{Comp}\langle E|S_2\rangle\sigma, & \text{otherwise}\end{cases}$$

$\mathbf{Comp}\langle E|S_1; S_2\rangle\sigma = \langle\sigma\rangle^\wedge\tau^\wedge\mathbf{Comp}\langle E|S_2\rangle(\kappa\tau)$, where $\tau = \mathbf{Comp}\langle E|S_1\rangle\sigma$.

In the sequel we will refer to this set of equations as CE, which is an abbreviation of "the Cook equations." Now there are some questions to be answered. Does there exist a function with the above properties? If so, is this function unique? We cannot provide the answers immediately because the above equations can be interpreted as a recursive definition which is not inductive.

Cook also was aware of these questions as the following quotation from [6] shows: "The definition is recursive, in the sense that **Comp** appears on the right side of the clauses. This may appear ironic in a paper on program verification, since one of the important issues in programming language semantics is interpreting recursively defined procedures. However, one does not have to understand recursive procedures in general in order to understand this specific definition. Suffice it to say that we intend **Comp** to be evaluated by "call by name," in the sense that occurrences of **Comp** are to be replaced successively by their meanings according to the appropriate clauses in the definition."

In this paper we will provide the answer to the above questions; there is a unique total function which satisfies the equations. We will show this in four different ways. The first idea is to derive from the recursive definition an inductive one which defines the elements of the outcome of **Comp** one by one. This is treated in § 2. The other techniques are based on a standard idea from denotational semantics: transform recursion into iteration. From the Cook equations an operator can be derived, and iteration of this operator yields a sequence of approximations which should tend to a limit, a function satisfying CE. In order to be able to talk about convergence, the relevant semantical domains are turned into cpo's.

However these techniques cannot be applied here straightforwardly, because the approximations will generally not converge. This phenomenon is analyzed in § 3. To make the basic idea work we have to extend the standard approach somehow and this can be done into three directions. First of all we can enrich the domain Σ^∞ by adding to it a class of finite rows marked as "not yet complete." Secondly, we can rephrase the Cook equations such that the standard approach does work. Lastly, we can make use of the fact that Σ^∞ has more structure than a cpo, it is a complete metric space. These solutions will be treated in § 4–6.

In the sequel we will need the following lemma which gives information on all total functions satisfying CE. The lemma states that a definition through a set of equations like CE is independent of the particular way we defined $\kappa\tau$ for $\tau = \langle\,\rangle$ or $\tau \in \Sigma^\omega$. This holds because CE is such that in it κ is never applied to $\langle\,\rangle$, and if κ is applied to an element of Σ^ω, then its value is irrelevant because it will be used only to determine a row which is appended to an infinite row, which means that it will be neglected.

LEMMA 1.1. *For every total function* Φ *in* $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ *which satisfies* CE the following holds.

1. *For all* R *and* σ *we have* $\Phi R\sigma \neq \langle\,\rangle$.

2. *If we construct a set of equations* CE′ *which is like* CE, *except for the fact that it uses another last element extraction function* κ' *which differs from* κ *only when applied to* $\langle\,\rangle$ *or elements from* Σ^ω, *then* Φ *is also a solution of* CE′.

2. A straightforward solution. The idea is the following. We define a new function **C** which is like **Comp** but takes besides R and σ an extra argument, a natural number n, and which yields an element from Σ. This element should then be the nth element of the row $\mathbf{Comp}R\sigma$. Now it is possible to give an inductive definition of **C**. First of all we have to introduce an extra element Ω ("undefined") because in the setup, as proposed here, it is possible to ask for the third element of a row of two elements. In such cases we then deliver Ω. We define

DEFINITION 2.1. The function $\mathbf{C}\colon \mathbf{Prog} \to \Sigma \to \mathbb{N} \to \Sigma \cup \{\Omega\}$ is defined by induction on n as follows:

$$\mathbf{C}\langle E|A\rangle\sigma n = \begin{cases} \mathbf{A}A\sigma & \text{if } n = 1, \\ \Omega & \text{otherwise;} \end{cases}$$

$$\mathbf{C}\langle E|P_i\rangle\sigma n = \begin{cases} \sigma & \text{if } n = 1, \\ \mathbf{C}\langle E|S_i\rangle\sigma(n-1) & \text{otherwise, where } P_i \Leftarrow S_i \text{ occurs in } E; \end{cases}$$

$$\mathbf{C}\langle E|\text{if } B \text{ then } S_1 \text{ else } S_2\rangle\sigma n$$
$$= \begin{cases} \sigma & \text{if } n = 1, \\ \mathbf{C}\langle E|S_1\rangle\sigma(n-1) & \text{if } n \neq 1 \text{ and } \mathbf{B}B\sigma = tt, \\ \mathbf{C}\langle E|S_2\rangle\sigma(n-1) & \text{otherwise;} \end{cases}$$

$$\mathbf{C}\langle E|S_1; S_2\rangle\sigma n$$
$$= \begin{cases} \sigma & \text{if } n = 1, \\ \mathbf{C}\langle E|S_1\rangle\sigma(n-1) & \text{if } n \neq 1 \text{ and } \mathbf{C}\langle E|S_1\rangle\sigma(n-1) \neq \Omega, \\ \mathbf{C}\langle E|S_2\rangle(\mathbf{C}\langle E|S_1\rangle\sigma k)(n-k-1) & \text{if } n \neq 1,\ \mathbf{C}\langle E|S_1\rangle\sigma(n-1) = \Omega, \\ & \text{and } V := \{m | \mathbf{C}\langle E|S_1\rangle\sigma m \neq \Omega \text{ and} \\ & \mathbf{C}\langle E|S_1\rangle\sigma(m+1) = \Omega \text{ and } m < n\} \neq \varnothing, \\ & \text{where } k = \min V, \\ \Omega, & \text{otherwise.} \end{cases}$$

Because we had to be careful about little details, the above definition has an awkward appearance. It can be made more tractable by realizing that for all R and σ the infinite row $\langle \mathbf{C}R\sigma k\rangle_k$ contains either only elements from Σ, or has the form $\langle \sigma_1, \sigma_2, \cdots, \sigma_k, \Omega, \Omega, \Omega, \cdots\rangle$. This observation enables us to rephrase the case $S_1; S_2$ in the above definition as follows.

LEMMA 2.2.

$$\mathbf{C}\langle E|S_1; S_2\rangle\sigma n = \begin{cases} \sigma & \textit{if } n = 1, \\ \mathbf{C}\langle E|S_1\rangle\sigma(n-1) & \textit{if } \mathbf{C}\langle E|S_1\rangle\sigma(n-1) \neq \Omega \textit{ and } n \neq 1, \\ \mathbf{C}\langle E|S_2\rangle(\mathbf{C}\langle E|S_1\rangle\sigma k)(n-k-1) & \textit{otherwise, where } k \textit{ is such that} \\ & \mathbf{C}\langle E|S_1\rangle\sigma k \neq \Omega \textit{ and} \\ & \mathbf{C}\langle E|S_1\rangle\sigma(k+1) = \Omega. \end{cases}$$

Now, the function **Comp** defined by

$$\mathbf{Comp}R\sigma = \begin{cases} \langle \mathbf{C}R\sigma 1, \cdots, \mathbf{C}R\sigma n\rangle & \text{if } \mathbf{C}R\sigma n \neq \Omega \text{ and } \mathbf{C}R\sigma(n+1) = \Omega, \\ \langle \mathbf{C}R\sigma 1, \mathbf{C}R\sigma 2, \cdots\rangle & \text{otherwise} \end{cases}$$

satisfies CE, as one can check straightforwardly.

Finally, we show that there is exactly one total function satisfying CE by the following argument. For any function **Comp** and for any R, σ and n we can calculate, using only the clauses from CE, the nth element from the row $\mathbf{Comp}R\sigma$, like we have done in Definition 2.1. So we have that the equations CE determine, for every R and σ, every element from the row $\mathbf{Comp}R\sigma$, that is, this row must be unique, that is **Comp** must be unique. Note that the above reasoning would no longer be valid if we allowed partial functions in $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ to be solutions of CE.

3. There is a problem if we try to use the fixed point approach. It is tempting to try to use fixed point theory to answer the questions raised in § 1, because any solution of CE will be a fixed point on the operator $\Psi: D \to D$, with $D = \mathbf{Prog} \to \Sigma \to \Sigma^\infty$ defined by

$$\begin{aligned} \Psi = \lambda\Phi.\lambda R.\lambda\sigma. & \\ & R \equiv \langle E|A\rangle \to \langle \mathbf{A}A\sigma\rangle, \\ & R \equiv \langle E|P_i\rangle \to \langle\sigma\rangle^\wedge \Phi\langle E|S_i\rangle\sigma, \\ & R \equiv \langle E|\text{if } B \text{ then } S_1 \text{ else } S_2\rangle \\ & \qquad \to (\mathbf{B}B\sigma = tt \to \langle\sigma\rangle^\wedge\Phi\langle E|S_1\rangle\sigma, \langle\sigma\rangle^\wedge\Phi\langle E|S_2\rangle\sigma), \\ & R \equiv \langle E|S_1, S_2\rangle \to \langle\sigma\rangle^\wedge\Phi\langle E|S_1\rangle\sigma^\wedge\Phi\langle E|S_2\rangle(\kappa(\Phi\langle E|S_1\rangle\sigma)). \end{aligned}$$

Now it is a well-known fact from denotational semantics ([12], [13], [14]; see also [15] or [3] which both give an introduction to the subject) that Ψ has a least fixed point $\mu\Psi$ if this operator is continuous. In that case $\mu\Psi$ equals the lub of the chain $\perp \sqsubseteq \Psi\perp \sqsubseteq \Psi(\Psi\perp) \sqsubseteq \cdots$.

So, if we manage to make D a cpo such that Ψ is continuous then we obtain the required existence result immediately. Again, it is well known that D is a cpo if there is an ordering $\sqsubseteq$ on Σ^∞ which makes this set a cpo. Now the intuition behind $\tau_1 \sqsubseteq \tau_2$ is that τ_2 contains more information than τ_1, or that τ_2 is a better approximation of some final result than τ_1. A technique for turning a set into a cpo that is often used is to make this set a flat cpo. That is, add a totally undefined element $\perp$ to it and define $\tau_1 \sqsubseteq \tau_2$ iff $\tau_1 = \tau_2$ or $\tau_1 = \perp$.

However, this construction is not suited for our purposes, because we obtain a least fixed point $\mu\Psi$ which yields the right result for terminating processes, but which yields $\perp$ for nonterminating processes. By way of an example we will evaluate some elements of the chain $\perp \sqsubseteq \Psi \perp \sqsubseteq \Psi^2 \perp \sqsubseteq \cdots$ approximating $\mu\Psi$, applied to the program $\langle P|P \Leftarrow P\rangle$:

1. $\perp\langle P|P\Leftarrow P\rangle\sigma = \perp$
2. $(\Psi\perp)\langle P|P\Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge \perp\langle P|P\Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge\perp = \perp$
3. $(\Psi^2\perp)\langle|P\Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge(\Psi\perp)\langle P|P\Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\perp\langle P|P\Leftarrow P\rangle\sigma$
   $= \langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\perp = \perp$
4. $(\Psi^3\perp)\langle P|P\Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge(\Psi^2\perp)\langle P|P\Leftarrow P\rangle\sigma$
   $= \langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\perp\langle P|P\Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\perp = \perp$ etc.

The problem is that the ordering in a flat cpo is not refined enough: an approximation τ_1 of a final answer τ ($\tau_1 \sqsubseteq \tau$) contains either all information ($\tau_1 = \tau$) or no information at all ($\tau_1 = \perp$). Now because all finite approximations of an infinite row are necessarily unequal to this row we must have that all these approximations are equal to $\perp$. That is we get a chain $\perp \sqsubseteq \perp \sqsubseteq \cdots$ with lub $\perp$ and this is not what we want.

This analysis also shows a way out. What the sequence of approximations given above should do is yield longer and longer initial segments of the final outcome. That is, the ordering should be such that $\langle\sigma\rangle \sqsubseteq \langle\sigma, \sigma\rangle \sqsubseteq \langle\sigma, \sigma, \sigma\rangle \sqsubseteq \cdots$ is a chain with the natural lub $\langle\sigma, \sigma, \sigma, \cdots\rangle$. This leads us to trying the prefix ordering on Σ^∞: $\tau_1 \sqsubseteq \tau_2$ iff τ_1 is a prefix of τ_2. One easily checks that Σ^∞ with this ordering is a cpo with the empty row $\langle\,\rangle$ as bottom element. This ordering yields a correct approximation sequence for the program $\langle P|P\Leftarrow P\rangle$ as one easily can check. However, this approach does not work in general because Ψ is not continuous under this ordering. This stems from the fact that the operators κ and $^\wedge$ are not continuous, not even monotonic under the prefix ordering. For instance, $\langle\sigma_1\rangle \sqsubseteq \langle\sigma_1, \sigma_2\rangle$ but $\kappa\langle\sigma_1\rangle = \sigma_1$ and $\kappa\langle\sigma_1, \sigma_2\rangle = \sigma_2$ might very well be incomparable.

We can also show in a less technical way that the new approach does not work. Consider the sequence $\langle \perp (= \lambda R.\lambda\sigma.\langle\,\rangle), \Psi\perp, \Psi^2\perp, \cdots\rangle$ and apply some of the elements thereof to the program $R = \langle E|P; A_2\rangle$ (where $E = \langle P \Leftarrow A_1\rangle$) and initial state σ. We get

$$\perp R\sigma = \langle\,\rangle, \qquad (\Psi\perp)R\sigma = \langle\sigma\rangle, \qquad (\Psi^2\perp)R\sigma = \langle\sigma, \sigma, \mathbf{A}A_2\sigma\rangle,$$

$$(\Psi^3\perp)R\sigma = \langle\sigma, \sigma, \mathbf{A}A_1\sigma, \mathbf{A}A_2(\mathbf{A}A_1\sigma)\rangle,$$

and it follows that $\Psi^2\perp \not\sqsubseteq \Psi^3\perp$. Therefore the prefix ordering on Σ^∞ is such that the sequence $\langle\perp, \Psi\perp, \Psi^2\perp, \cdots\rangle$ is not a chain, and thus Ψ cannot be continuous.

If we investigate what went wrong here, we see that in evaluating $(\Psi^2\perp)R\sigma$ we apply the last element function κ to a row of states which is not yet finished; that is, we start evaluating A_2 "too early," namely in a state σ which is not the final state resulting from evaluation of P. This analysis suggests two solutions for the difficulty. The first one is to enlarge Σ^∞ so that it contains also rows of states which are marked as "not yet completed" and to let the operators κ and $^\wedge$ act in the "right" (continuous) manner on these rows. Another possibility is to rewrite Ψ in such a way that it does not use the noncontinuous operators κ and $^\wedge$ any more. Finally we observe that, though the above approximation sequence is not a chain, the right outcome has been obtained in the end. This suggests that Ψ might be continuous if we would use a more subtle notion of continuity. The next three sections will be devoted to a discussion of these possibilities.

4. Adding unfinished rows to Σ^∞. We observed that in Σ^∞ finished and unfinished rows of states must be distinguished. We will arrange this as follows: a row $\langle\sigma_1, \cdots, \sigma_n\rangle$ will be marked as unfinished by adding the element $\perp$ to it, so that we get $\langle\sigma_1, \cdots, \sigma_n, \perp\rangle$. Notice that only finite rows can possibly be unfinished; infinite rows, which model nonterminating computations, cannot contain more information than they already do. The ordering $\langle\sigma_1,\cdots,\sigma_n,\perp\rangle\sqsubseteq\tau$ iff $\langle\sigma_1,\cdots,\sigma_n\rangle$ is a prefix of τ is natural. Furthermore, $\hat{}$ should not append its second argument if its first argument is an unfinished row. All this leads to the following list of definitions and properties.

DEFINITION 4.1.

1. $\Sigma^{\sim}=\Sigma^{*\perp}\cup\Sigma^{\infty}$, with Σ^{∞} as before, and where $\Sigma^{*\perp}$ is the set of all rows consisting of zero or more states followed by the symbol $\perp$.

2. For $\tau_1,\tau_2\in\Sigma^{\sim}$ we define $\tau_1\sqsubseteq\tau_2$ iff either $\tau_1=\tau_2$ or $\tau_1=\langle\sigma_1,\cdots,\sigma_n,\perp\rangle\in\Sigma^{*\perp}$ and $\langle\sigma_1,\cdots,\sigma_n\rangle$ is a prefix of τ_2.

3. $\Sigma_\perp=\Sigma\cup\{\perp\}$, the flat cpo derived from Σ.

4. $\kappa:\Sigma^{\sim}\to\Sigma_\perp$ is defined by

$$\kappa(\tau)=\begin{cases}\perp & \text{if } \tau\in\Sigma^{\infty},\ \tau\in\Sigma^{*\perp} \text{ or } \tau=\langle\,\rangle,\\ \sigma_n & \text{if } \tau=\langle\sigma_1,\cdots,\sigma_n\rangle\in\Sigma^*\backslash\{\langle\,\rangle\}.\end{cases}$$

5. $\hat{}:\Sigma^{\sim}\times\Sigma^{\sim}\to\Sigma^{\sim}$ is defined by

$$\tau_1\hat{}\tau_2=\begin{cases}\tau_1 \quad \text{if } \tau_1\in\Sigma^{\omega}\cup\Sigma^{*\perp},\\ \langle\sigma_1,\cdots,\sigma_n,\sigma_1',\cdots,\sigma_k'\rangle \quad \text{if } \tau_1=\langle\sigma_1,\cdots,\sigma_n\rangle\in\Sigma^*,\ \tau_2=\langle\sigma_1',\cdots,\sigma_k'\rangle\in\Sigma^*,\\ \langle\sigma_1,\cdots,\sigma_n,\sigma_1',\cdots,\sigma_k',\perp\rangle\\ \qquad\qquad \text{if } \tau_1=\langle\sigma_1,\cdots,\sigma_n\rangle\in\Sigma^*,\ \tau_2=\langle\sigma_1',\cdots,\sigma_k',\perp\rangle\in\Sigma^{*\perp},\\ \langle\sigma_1,\cdots,\sigma_n,\sigma_1',\cdots\rangle \quad \text{if } \tau_1=\langle\sigma_1,\cdots,\sigma_n\rangle\in\Sigma^*,\ \tau_2=\langle\sigma_1',\cdots\rangle\in\Sigma^{\omega}.\end{cases}$$

LEMMA 4.2.

1. $(\Sigma^{\sim},\sqsubseteq)$ *is a cpo with smallest element* $\langle\perp\rangle$.

2. κ *and* $\hat{}$ *are continuous.*

Now that we have added the element $\perp$ to Σ we have to adapt the definition of a little bit.

DEFINITION 4.3. $\Psi:D\to D$, with $D=\mathbf{Prog}\to\Sigma_\perp\to_s\Sigma^{\sim}$ is defined by

$$\begin{aligned}\Psi=\lambda\Phi.\lambda R.\lambda\sigma.\sigma&=\perp\to\langle\perp\rangle,\\ R&\equiv\langle E|A\rangle\to\langle\mathbf{A}A\sigma\rangle,\\ R&\equiv\langle E|P_i\rangle\to\langle\sigma\rangle\hat{}\Phi\langle E|S_i\rangle\sigma,\\ R&\equiv\langle E|\text{ if } B \text{ then } S_1 \text{ else } S_2\rangle\\ &\quad\to(\mathbf{B}B\sigma=tt\to\langle\sigma\rangle\hat{}\Phi\langle E|S_i\rangle\sigma,\langle\sigma\rangle\hat{}\Phi\langle E|S_2\rangle\sigma),\\ R&\equiv\langle E|S_1;S_2\rangle\to\langle\sigma\rangle\hat{}\Phi\langle E|S_1\rangle\sigma\hat{}\Phi\langle E|S_2\rangle(\kappa(\Phi\langle E|S_1\rangle\sigma)).\end{aligned}$$

Remarks.

1. The expression $\Sigma_\perp\to_s\Sigma^{\sim}$ denotes the cpo of all strict functions from $\Sigma_\perp$ to $\Sigma^{\sim}$, that is, all functions f for which $f\perp=\langle\perp\rangle$. This precaution is needed because otherwise Ψ would not be continuous.

2. One easily checks that the operator Ψ has the functionality as announced. That is, for all Φ in $\mathbf{Prog}\to\Sigma_\perp\to_s\Sigma^{\sim}$, $R\in\mathbf{Prog}$ and $\sigma\in\Sigma_\perp$, we have that $\Psi\Phi R\sigma\in\Sigma^{\sim}$ (that is, only the last element might be $\perp$); and also for all $\Phi\in D$, $R\in\mathbf{Prog}$ we have that $\Psi\Phi R\perp=\langle\perp\rangle$ (i.e. $\Psi\Phi R$ is strict again).

3. The fact that Ψ is a continuous operator in $D\to D$ and thus that $\mu\Psi$ exists, can proved straightforwardly.

The key lemma is

LEMMA 4.4. *For all R and $\sigma\neq\perp$ we have* $(\mu\Psi)R\sigma\in\Sigma^{\infty}$.

Proof. By contradiction. Suppose the assertion is not true. We then would have some R and $\sigma \neq \perp$ for which $(\mu\Psi)R\sigma \in \Sigma^{*\perp}$. Now $(\mu\Psi)R\sigma = \bigsqcup_i ((\Psi^i \perp)R\sigma)$ and therefore we would have that for all i·, $\tau_i := (\Psi^i \perp)R\sigma \in \Sigma^{*\perp}$. Now intuitively $\tau_i \in \Sigma^{*\perp}$ means that this approximation of evaluation of R in σ is not good enough, because this row is not yet completed. This suggests that there is a better approximation in the chain $\langle(\Psi^i \perp)R\sigma\rangle_i$ and in fact this holds already for the next element in the chain: we have $\tau_i \in \Sigma^{*\perp} \Rightarrow \tau_{i+1} \neq \tau_i$ (to be proved by induction on i). Thus we have the following situation: $(\mu\Psi)R\sigma$ is the lub of a strictly increasing chain $\perp R\sigma \sqsubsetneq (\Psi\perp)R\sigma \sqsubsetneq \cdots$ with all $(\Psi^k \perp)R\sigma \in \Sigma^{*\perp}$. Now we have a contradiction, for such a chain must have a lub in Σ^ω.

THEOREM 4.5. *$\mu\Psi$, restricted to the domain* $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$, *is the unique solution of* CE.

Proof.

1. Notice that we cannot state that $\mu\Psi$ is a solution of CE, because $\mu\Psi$ is an element of $\mathbf{Prog} \to \Sigma_\perp \to_s \Sigma^\sim$ and as such it can never be a solution of CE. Notice also that we can restrict $\mu\Psi$ to the domain $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ only by virtue of Lemma 4.4.

2. [$\mu\Psi$ is a solution.] First compare the definition of κ and $\hat{}$ from § 1 with the ones in Definition 4.1 and observe that the restriction of $\hat{}$ (according to 4.1.5) to $\Sigma^\infty \times \Sigma^\infty$ is the same operator as $\hat{}$ in § 1, while the restriction of κ to Σ^∞ is almost the same, the only difference being the cases $\kappa\tau$ where $\tau \in \Sigma^\omega$ or $\tau = \langle\,\rangle$. If these operators would be the same then we were ready, because from Definition 4.3 we see that κ and $\hat{}$ are applied only to arguments of the form $(\mu\Psi)R\sigma$ and these are in Σ^∞ by Lemma 4.4. However the values of $\kappa\langle\,\rangle$ and $\kappa\tau$ for $\tau \in \Sigma^\omega$ are irrelevant, because the fixed points of Ψ have the same properties as the ones given by Lemma 1.1 for the solutions of CE.

3. [$\mu\Psi$ is the only fixed point Ψ.] Suppose not. Then there would be a bigger fixed point Φ, that is, there would be an R and σ such that $(\mu\Psi)R\sigma \sqsubsetneq \Phi R\sigma$. This is impossible, however, because by Lemma 4.4 $(\mu\Psi)R\sigma \in \Sigma^\infty$ which means that $(\mu\Psi)R\sigma$ is a maximal element in $\Sigma^\sim$.

4. [$\mu\Psi$ is the only solution of CE.] Suppose there would be another function $\mathbf{C}: \mathbf{Prog} \to \Sigma \to \Sigma^\infty$ satisfying CE. We can extend this function to a function $\mathbf{C}': \mathbf{Prog} \to \Sigma_\perp \to_s \Sigma^\sim$ by defining $\mathbf{C}'R\sigma = \mathbf{C}R\sigma$ if $\sigma \in \Sigma$ and $\langle\perp\rangle$ if $\sigma = \perp$. One easily checks that $\mathbf{C}'$ is a fixed point of Ψ, but then $\mathbf{C}' = \mu\Psi$, a contradiction.

5. The continuation approach. In § 3 we remarked that the direct fixed point approach failed due to the fact that the operators κ and $\hat{}$ are not continuous. In this section we will find a way out of this problem by restructing CE in such a way that these operators are not used any more, or at least not in a noncontinuous way. The problem stems from the clause on constructs of the form $\langle E|S_1; S_2\rangle$. The idea that we will pursue is to use continuation semantics instead of direct semantics.

Direct semantics defines the meaning of a construct in terms of the rows of states that correspond to evaluation of the constituents of the construct. Therefore the operators κ and $\hat{}$ have to be used: the meaning of $\langle E|S_1; S_2\rangle$ is obtained by concatenating the rows of states corresponding to the meanings of $\langle E|S_1\rangle$ and $\langle E|S_2\rangle$. Continuation semantics uses another idea: the meaning of a construct is the row of states which is the result of evaluating the construct itself followed by evaluation of the rest of the program of which the construct is supposed to be a part. Of course, the effect of evaluation of the rest of the program cannot be obtained from the construct itself; so we have to give the meaning function another argument, a continuation which will be a function from states to rows of states describing the effect of the rest of the program. One can view this continuation as a coding of the row of statements which are to be evaluated once the statement under consideration has been worked through. More information on continuation semantics can be found in [15].

In this setup we do not have to concatenate two rows any more while defining the meaning of $\langle E|S_1; S_2\rangle$ because the effect of evaluating S_2 can be caught by changing the continuation which describes what will happen once the whole construct has been evaluated into a continuation which describes the effect of first evaluating S_2 and then applying the original continuation. This new formed continuation is given as an argument to $\mathbf{Comp}\langle E|S_1\rangle$. All this leads to the following operator.

DEFINITION 5.1. The operator $\Psi: D \to D$, with $D = \mathbf{Prog} \to [\Theta \to \Theta]$ and $\Theta = \Sigma \to \Sigma^\infty$ is defined by

$$\Psi = \lambda\Phi.\lambda R.\lambda\theta.\lambda\sigma.$$

$$R \equiv \langle E|A\rangle \to \langle \mathbf{A}A\sigma\rangle^\wedge\theta(\mathbf{A}A\sigma),$$

$$R \equiv \langle E|P_i\rangle \to \langle\sigma\rangle^\wedge\Phi\langle E|S_i\rangle\theta\sigma,$$

$$R \equiv \langle E|\text{if } B \text{ then } S_1 \text{ else } S_2\rangle$$

$$\to (\mathbf{B}B\sigma = tt \to \langle\sigma\rangle^\wedge\Phi\langle E|S_1\rangle\theta\sigma, \langle\sigma\rangle^\wedge\Phi\langle E|S_2\rangle\theta\sigma),$$

$$R \equiv \langle E|S_1; S_2\rangle \to \langle\sigma\rangle^\wedge\Phi\langle E|S_1\rangle\{\Phi\langle E|S_2\rangle\theta\}\sigma.$$

Remarks.

1. Notice that the operator κ is not used any more. We do use the concatenation operator, but only in a continuous way: $\lambda\tau.\langle\sigma\rangle^\wedge\tau$ is continuous with respect to the prefix order on Σ^∞.

2. The fourth clause of the definition can be interpreted as follows: evaluating $\langle E|S_1; S_2\rangle$ followed by evaluation according to θ amounts to evaluation of $\langle E|S_1\rangle$ followed by [evaluation of $\langle E|S_2\rangle$ followed by evaluating according to θ].

3. The domain $[\Theta \to \Theta]$ is the cpo of all continuous functions from Θ to Θ.

4. Ψ is well defined, in the sense that for all $\Phi \in D$ we have $\Psi\Phi \in D$, or in other words: $\forall\Phi \in D\ \forall R \in \mathbf{Prog}\ \forall\theta_1 \sqsubseteq \theta_2 \sqsubseteq \cdots : \Psi\Phi R(\bigsqcup_i \theta_i) = \bigsqcup_i \Psi\Phi R\theta_i$.

5. Ψ is continuous and therefore $\mu\Psi$ exists (notice that $D = \mathbf{Prog} \to \Theta \to \Theta$ would not work).

We now define $\mathbf{Comp} = \lambda R.\lambda\sigma\ (\mu\Psi)R\{\lambda\sigma.\langle\,\rangle\}\sigma$, and the next thing to prove is that this function is a solution of CE. The proof is by cases, and the only nontrivial case is to prove that

$$(*) \qquad \mathbf{Comp}\langle E|S_1; S_2\rangle\sigma = \langle\sigma\rangle^\wedge\mathbf{Comp}\langle E|S_1\rangle\sigma^\wedge\mathbf{Comp}\langle E|S_2\rangle\sigma'$$

with σ' as usual. Now

$$\mathbf{Comp}\langle E|S_1; S_2\rangle\sigma = (\mu\Psi)\langle E|S_1; S_2\rangle\{\lambda\sigma.\langle\,\rangle\}\sigma$$

$$= \langle\sigma\rangle^\wedge(\mu\Psi)\langle E|S_1\rangle\{(\mu\Psi)\langle E|S_2\rangle\{\lambda\sigma.\langle\,\rangle\}\}\sigma,$$

and the right-hand side of $(*)$ equals

$$\langle\sigma\rangle^\wedge(\mu\Psi)\langle E|S_1\rangle\{\lambda\sigma.\langle\,\rangle\}\sigma^\wedge(\mu\Psi)\langle E|S_2\rangle\{\lambda\sigma.\langle\,\rangle\}\sigma',$$

where $\sigma' = \kappa((\mu\Psi)\langle E|S_1\rangle\{\lambda\sigma.\langle\,\rangle\}\sigma)$.

We thus have to establish a correspondence between the old definition of composition which used κ and $^\wedge$ and the new one which uses continuations. This correspondence is phrased in the next "continuation removal" lemma, which must be clear if the idea behind continuations has been well understood.

LEMMA 5.2. *Let $\Phi \in D = \mathbf{Prog} \to [\Theta \to \Theta]$ be a fixed point of Ψ. For all R, θ and σ we have that $\Phi R\theta\sigma = \tau^\wedge\theta(\kappa\tau)$, where $\tau = \Phi R\{\lambda\sigma.\langle\,\rangle\}\sigma$.*

Proof. Two cases.

1. τ is infinite. Then $\tau\,\hat{}\,\theta(\kappa\tau)=\tau$. On the other hand, Φ is continuous in θ and thus monotonic. This means $\tau=\Phi R\{\lambda\sigma.\langle\,\rangle\}\sigma\sqsubseteq\Phi R\theta\sigma$. But τ, being infinite, is maximal, and therefore $\tau=\Phi R\theta\sigma$.

2. The case that τ is finite can be proved by induction on the length of τ.

THEOREM 5.3. *The function* **Comp** *as defined in this section is the unique solution of* CE.

Proof.

1. That **Comp** is a solution of CE follows from the remarks preceding Lemma 5.2.

2. We now prove that Ψ has exactly one fixed point. Let $\Phi\in\mathbf{Prog}\to[\Theta\to\Theta]$ be a fixed point of Ψ such that $\mu\Psi\sqsubseteq\Phi$. We can prove that for all R, θ and σ we have $(\mu\Psi)R\theta\sigma=\Phi R\theta\sigma$.

a. If $(\mu\Psi)R\theta\sigma$ is infinite then it is maximal in Σ^∞. The desired equality then follows from $(\mu\Psi)R\theta\sigma\sqsubseteq\Phi R\theta\sigma$.

b. For all finite $(\mu\Psi)R\theta\sigma$ the desired equality can be proved by induction on its length.

3. For every solution $\mathbf{C}$ of CE we define $\alpha\mathbf{C}:=\lambda R.\lambda\theta.\lambda\sigma.\mathbf{C}R\sigma\,\hat{}\,\theta(\kappa(\mathbf{C}R\sigma))$ (compare Lemma 5.2). We can show in a straightforward way that every such $\alpha\mathbf{C}$ is a fixed point of Ψ. Notice that $\alpha\mathbf{C}\in\mathbf{Prog}\to[\Theta\to\Theta]$ must hold, i.e. $\alpha\mathbf{C}$ must be continuous in its continuation parameter.

4. Suppose CE has more than one solution say $\mathbf{C}$ and $\mathbf{C}'$. Then there exist R and σ such that $\mathbf{C}R\sigma\neq\mathbf{C}'R\sigma$. But then $\alpha\mathbf{C}$ and $\alpha\mathbf{C}'$ are both fixed points of Ψ, with $(\alpha\mathbf{C})R\theta\sigma\neq(\alpha\mathbf{C}')R\theta\sigma$, which contradicts 2.

6. Σ^∞ as a metric topological space.

At the end of § 3 we investigated the approximation sequence $\perp R\sigma$, $(\Psi\perp)R\sigma$, $(\Psi^2\perp)R\sigma$, $(\Psi^3\perp)R\sigma$, with $R=\langle P\Leftarrow A_1|P;A_2\rangle$ and Ψ the operator derived from CE. We observed that this sequence was not a chain though it converged (in some sense) to the right result. This phenomenon also holds for nonterminating computations like the evaluation of $\langle P\Leftarrow A_1;P|P;A_2\rangle$ in some σ. We can prove that the above observations hold in general, the key lemma is the following.

LEMMA 6.1. *If Φ_1,Φ_2 are such that for all R and σ the sequences $\Phi_1R\sigma$ and $\Phi_2R\sigma$ agree on their first n places, then for all R and σ we have that $(\Psi\Phi_1)R\sigma$ and $(\Psi\Phi_2)R\sigma$ agree on at least their first $n+1$ places.*

Proof. Straightforward by cases (if the sequence τ_1 or τ_2 has length smaller than k, we have by definition that τ_1 and τ_2 agree on their first k places iff $\tau_1=\tau_2$).

From this lemma we can deduce that for $n>m$, we have that $(\Psi^n\perp)R\sigma$ and $(\Psi^m\perp)R\sigma$ agree on at least their first m elements. Therefore we can define $\lim\langle(\Psi^k\perp)R\sigma\rangle_k$ as the sequence in Σ^∞ that agrees for every n on its first n elements with $(\Psi^n\perp)R\sigma$. Though we have not defined exactly what "convergence" means, it must be clear that, informally, the sequence $\langle(\Psi^k\perp)R\sigma\rangle_k$ converges to this limit. This convergence is uniform in R and σ in the sense that for all R and σ the first n elements in $(\Psi^n\perp)R\sigma$ are "correct."

If we define $\lim(\Psi^k\perp)$ as $\lambda R.\lambda\sigma.\lim\langle(\Psi^k\perp)R\sigma\rangle_k$ we have that $\Psi(\lim(\Psi^k\perp))=\lim(\Psi^k\perp)$. This holds, because we can prove by induction on n that for all R, σ and n the rows $\Psi(\lim\Psi^k\perp)R\sigma$ and $\lim(\Psi^k\perp)R\sigma$ agree on their first n elements (i.e. the first n elements of $(\Psi^n\perp)R\sigma$). Finally, we can show that Ψ has not more than one fixed point by the following argument. Suppose that there are two fixed points Φ_1 and Φ_2. We prove that for all R, σ the rows $\Phi_1R\sigma$ and $\Phi_2R\sigma$ agree on their first n elements (by induction on n). The case $n=0$ is immediate. If $\Phi_1R\sigma$ and $\Phi_2R\sigma$ agree on $n-1$ elements for all R and σ, then by Lemma 6.1 $(\Psi\Phi_1)R\sigma$ and $(\Psi\Phi_2)R\sigma$ agree on n elements. But Φ_1 and Φ_2 are fixed points and thus we have that $(\Psi\Phi_i)R\sigma = \Phi_i R\sigma$ for $i = 1, 2$.

We can rephrase all this in the language of topology, by defining that τ_1, τ_2 are close to each other if they have a big common prefix (viz. Definition 6.2). This makes Σ^∞ a metric space and it is possible to formalize the above argument in terms of these topological notions. However, from Lemma 6.1, we can easily derive that the operator Ψ is a contraction (Lemma 6.8), and this means that our fixed point result can be derived in a more elegant way; it is equivalent to a well known theorem from topology. This approach is inspired by an endeavor to apply Nivat's results (see, for example, [11]) to the problem treated in this paper. We saw no way to achieve this, but the basic facts about Σ^∞ that he provided were very useful. In fact, the whole treatment given in this chapter is much in the style of Nivat's.

DEFINITION 6.2.

1. We denote, for $\tau \in \Sigma^\infty$ by $\tau[n]$ the prefix of τ consisting of the first n elements of τ, or τ itself if its length is smaller than n.
2. We define the following distance function d on Σ^∞:

$$d(\tau_1, \tau_2) = \begin{cases} 2^{-n} & \text{if } \tau_1[n-1] = \tau_2[n-1] \text{ and } \tau_1[n] \neq \tau_2[n], \\ 0 & \text{otherwise.} \end{cases}$$

LEMMA 6.3. *d is a metric, i.e. we have the familiar properties:*

$$d(\tau_1, \tau_2) = 0 \textit{ iff } \tau_1 = \tau_2,$$
$$d(\tau_1, \tau_2) = d(\tau_2, \tau_1),$$
$$d(\tau_1, \tau_2) \leqq d(\tau_1, \tau_3) + d(\tau_2, \tau_3).$$

Now the metric space (Σ^∞, d) is complete.

LEMMA 6.4. *Every Cauchy sequence $\langle \tau_i \rangle_i$ in Σ^∞ converges.*

Proof [11]. For every k there is an $N(k)$ such that $d(\tau_n, \tau_m) < 2^{-k}$ for all $n, m \geqq N(k)$. Define $\tau_{(k)} = \tau_{N(k)}[k]$. Then $\tau_{(k)}$ agrees on its first k elements with every τ_n for $n \geqq N(k)$. The sequence $\langle \tau_{(k)} \rangle_k$ is a chain, say with lub τ. Now $\langle \tau_i \rangle_i$ converges to τ.

The next thing to do is to make $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ a metric space by defining

DEFINITION 6.5. $d'(\Phi_1, \Phi_2) = \text{lub}\,\{d(\Phi_1 R\sigma, \Phi_2 R\sigma) \mid R \in \mathbf{Prog}, \sigma \in \Sigma\}$. We have that $(\mathbf{Prog} \to \Sigma \to \Sigma^\infty, d')$ is a complete metric space too.

LEMMA 6.6. *The function d' is a metric, and every Cauchy sequence $\langle \Phi_k \rangle_k$ in $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ converges with limit $\lambda R.\lambda\sigma.\text{lim}\, \Phi_k R\sigma$.*

Proof. Standard topology.

LEMMA 6.7. *If for all R, σ we have $d(\Phi_1 R\sigma, \Phi_2 R\sigma) \leqq 2^{-n}$, then for all R, σ: $d((\Psi\Phi_1)R\sigma, (\Psi\Phi_2)R\sigma) \leqq 2^{n-1}$.*

Proof. This is Lemma 6.1.

LEMMA 6.8. *Ψ is a contraction, in particular we have that for all Φ_1, Φ_2*

$$d'(\Psi\Phi_1, \Psi\Phi_2) \leqq \tfrac{1}{2} d'(\Phi_1, \Phi_2).$$

Proof. Follows immediately from Lemma 6.7.

THEOREM 6.9. *Ψ has exactly one fixed point.*

Proof. This is the contraction mapping theorem, viz. [5], [8].

7. Concluding remarks. In a certain sense we have worked in a direction opposite to the one Scott took when he devised his theory of computing. He wanted to exploit notions from topology such as limit and continuity, and therefore he introduced cpo's because the domains on which programs compute are in general not of a topological kind. We found in § 3 that Σ^∞ considered as a cpo did not have enough structure to prove the desired result. However by using the inherent topology on Σ^∞ we were able to derive this result in an elegant manner (§ 6).

The above results have been derived for a rather simple paradigm language, but the techniques used here can be applied to more sophisticated languages, in particular the language used in Cook's paper [6].

The theory as it stands now cannot be applied to nondeterministic programs, and, as a consequence of this, neither to parallel programs. This is due to the fact that nondeterministic programs generate trees and not rows. However, it seems that the techniques presented here can be extended to trees as well. Part of this extension is reported on in [9].

The central theorem that we have proved four times in this paper holds also if the Cook equations have expressions in their right-hand sides which do not start with a constant one element row. Notice that we have to be careful here. For instance we cannot leave out the $\langle\sigma\rangle$ in the second clause on procedure calls in CE (§ 1) because if we had done so, then $\mathbf{Comp}\langle P \Leftarrow P|P\rangle\sigma$ would not yield an infinite row, which it should do because $\langle P \Rightarrow P|P\rangle$ specifies a nonterminating computation.

Let us investigate the consequences of changing CE such that the fourth clause is altered into

$$\mathbf{Comp}\langle E|S_1; S_2\rangle\sigma = \mathbf{Comp}\langle E|S_1\rangle\sigma\,{}^\wedge\mathbf{Comp}\langle E|S_2\rangle(\kappa(\mathbf{Comp}\langle E|S_1\rangle\sigma)).$$

The central theorem of this paper would then be much harder to prove. For instance Definition 2.1 must now be by induction on $\langle n, \text{length}\,(R)\rangle$ instead of n, and the same holds for induction arguments in some other proofs (for instance Lemma 5.2). Furthermore, the statement $\tau_i \in \Sigma^{*\perp} \Rightarrow \tau_{i+1} \neq \tau_i$ in the proof of Lemma 4.4 is no longer true, as the counterexample $R \equiv \langle E|A_1; A_2\rangle$ and $i = 0$ shows. A weaker version of the lemma holds through:

$$\forall R, \sigma \neq \perp \quad \exists k : \Phi_i R\sigma \in \Sigma^{*\perp} \Rightarrow \Phi_{i+k} R\sigma \neq \Phi_i R\sigma.$$

In § 6 the central Lemma 6.1 does not hold any more, and the sequences $\langle \Psi^k \Phi\rangle_k$ are no longer uniformly convergent (for arbitrary Φ) in R and σ. We have to approach the problem differently. We cannot use the lub distance on $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ any more, but we have to use the pointwise extension of convergence in Σ^∞, quite analogously to how theory has been set up for cpo's. We now give a brief sketch of how the theorem can be deduced under these new circumstances.

1. Definition. $\langle \Phi_k\rangle_k$ converges iff $\forall R, \sigma : \langle \Phi_k R\sigma\rangle_k$ converges. In that case we define $\lim \Phi_k$ as $\lambda R.\lambda\sigma.(\lim \Phi_k R\sigma)$.

2. Lemma. *Ψ is continuous, in the sense that for all converging sequences $\langle \Phi_k\rangle_k$ we have* $\lim \Psi\Phi_k = \Psi(\lim \Phi_k)$.

3. Lemma. $\forall R, \sigma, n\ \exists N : k \geqq N \Rightarrow \forall \Phi_1, \Phi_2 : d((\Psi^k\Phi_1)R\sigma, (\Psi^k\Phi_2)R\sigma) \leqq 2^{-n}$.

This is a useful lemma, in some sense the analogue of Lemma 6.7. Notice that the N in the lemma is in general dependent on R and σ. The proof is by induction on the entity $\langle n, \text{length}\,(R)\rangle$. The lemma has the following useful consequences (4 and 5).

4. Lemma. *For all Φ we have that $\langle \Psi^k\Phi\rangle_k$ converges.*

5. Lemma. *The limit of $\langle \Psi^k\Phi\rangle_k$ is independent of the initial value Φ.*

6. Theorem. *The (changed) Cook equations have exactly one solution.*

Proof. There is a fixed point (for instance $\lim (\Psi^k\perp) =: \mu\Psi$), by results 2 and 4. If there was another fixed point Φ_0, then we would have that $\mu\Psi = \lim \Psi^k\Phi_0 = \lim \langle \Phi_0, \Phi_0, \cdots\rangle = \Phi_0$ (the first equality holds by result 5).

The above remarks show that it pays off (technically) to demand that the right-hand sides in CE all begin with a constant row. There are other reasons for this. The operational semantics yields a row of states which is intended as the trace left by execution of the program under consideration. Now execution of (for instance) $A_1; A_2$ can be divided into three parts: namely first determining that the statement is a composition of two other statements, secondly evaluating the first statement, and lastly evaluating the second one. It is reasonable that each stage of this evaluation has its effect on the trace. More generally, every clause in the Cook equations should add an element to the trace because it corresponds either to some elementary action, or to a decomposition of the statement being evaluated.

Related work and acknowledgments. In a letter to Cook [1], Krzysztof Apt suggested a method to compute **Comp** which is related to the technique of § 2: he proposes to define by induction on k the row $\mathbf{Comp}'R\sigma k$ which should consist of the first k elements of $\mathbf{Comp}R\sigma$. Having defined **Comp**$'$ he then defines $\mathbf{Comp}R\sigma = \tau$ iff $\exists k: \mathbf{Comp}'R\sigma n = \tau$ for all $n \geqq k$. He therefore defines **Comp** only for finite rows. The same holds for the results of Jeff Zucker in the appendix of [3]. He defines **Comp** as a fixed point of a set of equations derived from CE. He does this by using the recursion theorem. The technique in § 4 of adding the bottom element $\perp$ to mark a row as not yet completed has been used by Ralph Back in his analysis of unbounded nondeterminism [2]. The results in § 6 were inspired by the reading of Nivat's and others work on infinite computations, as reported on for instance in [11]. The topology on Σ^∞ was presented there, and also the proof of Lemma 6.4 can be found there.

A more elaborate version of this paper (more remarks and better worked out proofs) is registered as Mathematical Centre Report [4].

I acknowledge with pleasure the assistance of the following persons: the members of the Dutch working group on semantics, in particular Ruurd Kuiper with whom I had frequent and stimulating discussions on the material presented here. I would like to thank Jaco de Bakker and the referees for useful comments on the manuscript, and finally I would also like to thank Nizethe Kemmink and Susan Carolan for doing such a good typing job on a rather disjointed manuscript.

REFERENCES

[1] K. R. APT, personal communication.

[2] R. J. BACK, *Semantics of unbounded nondeterminism*, in Proc. 7th Colloquim on Automata, Languages and Programming, J. W. de Bakker and J. van Leeuwen, eds., Lecture Notes in Computer Science 85, Springer, New York, 1980, pp. 51–63.

[3] J. W. DE BAKKER, *Mathematical Theory of Program Correctness*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[4] A. DE BRUIN, *On the existence of Cook semantics*, Report IW 163/81, Mathematical Centre, Amsterdam, 1981.

[5] G. CHOQUET, *Cours d'analyse, Tôme* II, *Topologie*, Masson and Cie, Paris, 1964.

[6] S. A. COOK, *Soundness and completeness of an axiom system for program verification*, this Journal, 7 (1978), pp. 70–90.

[7] C. A. R. HOARE AND P. E. LAUER, *Consistent and complementary formal theories of the semantics of programming languages*, Acta Informatica, 3 (1974), pp. 135–153.

[8] A. N. KOLMOGOROV AND S. V. FOMIN, *Elements of the Theory of Functions and Functional Analysis*, Graylock Press, Rochester, NY, 1957.

[9] R. KUIPER, *An operational semantics for nondeterminism equivalent to a denotational one*, in Proc. International Symposium on Algorithmic Languages, J. W. de Bakker and J. C. van Vliet, eds., North-Holland, Amsterdam, 1981, pp. 373–398.

[10] P. E. LAUER, *Consistent formal theories of the semantics of programming languages*, IBM Laboratory Vienna, Techn. Report TR 25-121, 1971.

[11] M. NIVAT, *Infinite words, infinite trees, infinite computations*, in Foundations of Computer Science III, Part 2: Languages, Logic, Semantics, J. W. de Bakker and J. van Leeuwen, eds., Mathematical Centre Tracts 109, Mathematical Centre, Amsterdam, pp. 1–52.

[12] D. SCOTT, *Outline of a mathematical theory of computation*, Technical monograph PRG-2, Oxford University Computing Laboratory, Programming Research Group, 1970.

[13] ———, *The lattice of flow diagrams*, Technical monograph PRG-3, Oxford University Computing Laboratory, Programming Research Group, 1971.

[14] ———, *Continuous lattices*, Technical monograph PRG-7, Oxford University Computing Laboratory, Programming Research Group, 1971.

[15] J. E. STOY, *Denotational Semantics—The Scott–Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, 1977.

chapter 3

OPERATIONAL AND DENOTATIONAL SEMANTICS DESCRIBING THE MATCHING PROCESS IN SNOBOL4

This paper is registered as Report IW 151/80, Mathematical Centre, Amsterdam (1980)

Operational and denotational semantics describing the matching process in SNOBOL4

by

A. de Bruin

ABSTRACT

The pattern matching process in SNOBOL4 is investigated. We consider a subset of the language which is simple in this respect that patterns are not allowed as values of variables. This leads to matching processes that always terminate. After an informal description of the matching algorithm we present an operational semantics in the SECD-machine style. This semantics uses a stack to implement the backtracking which can occur during matching. After that a denotational semantics is introduced which uses continuations to describe the backtracking. Equivalence of the two semantics is then proved. The operational and denotational semantics are as similar as possible while retaining the typical operational respectively denotational ideas. This leads to a straightforward equivalence proof. That this similarity pays off is shown by another, somewhat disparate operational semantics for which equivalence with the denotational semantics is much harder to prove.

KEY WORDS & PHRASES: *Denotational semantics, operational semantics, continuation, backtracking, pattern matching, SNOBOL4, SECD machines*

1. INFORMAL DESCRIPTION OF THE SNOBOL4 FRAGMENT

This chapter serves as an introduction for those who are not acquainted with SNOBOL4, or only superficially so. We will discuss here only that part of the language which will be dealt with in the rest of the paper. Details and differences with full SNOBOL4 will be discussed in later chapters. The reader who is familiar with the language can therefore skip this chapter.

Pattern matching is in essence investigating whether a string of symbols (the *subject string*) is of a certain form as specified by a *pattern*. Another entity which has a role in this process is the *cursor*, a variable with integer values between zero and the length of the subject string, which serves as a pointer into this string. Cursor = 0 means that the pointer is located at the beginning of the string, the cursor being equal to the length of the subject string corresponds to the pointer being placed at the right-hand end of the string. In general, cursor = n denotes that the pointer is positioned between the n-th and the $(n+1)$-th symbol in the subject string.

Matching a string h against a pattern p starting with cursor position n may have two outcomes. The match may fail, which means that the substring of h starting directly after the n-th symbol does not have the form prescribed by p. The alternative is that the match succeeds in which case the process will yield a new cursor value, for instance n'. This will happen if the substring of h between cursor position n and n' has the property specified by p.

We will now present several forms which patterns can take, and explain their meaning. The first possibility is that a pattern is a string, for instance h'. Matching the subject h against the pattern h' succeeds if $h \equiv h_1h'h_2$ and if the cursor is located just before h'. After the match the cursor will then be placed just before the substring h_2 in the subject. In all other cases the match will fail. For example, if the subject string is 'arie' and the pattern is 'ri', then the match fails if the precursor position (the value of the cursor before the match) equals 0, and the match will succeed if the precursor position is 1. In the latter case the corresponding postcursor value will be 3.

The pattern <u>*nil*</u> matches every string without altering the cursor position. Matching against this pattern is the same as matching against the pattern formed by the empty string. The pattern *fail* is a pattern which fails invariably whatever the subject and the cursor position might be.

There are two dyadic operators on patterns, namely $\vee$ and &. The pattern $p_1 \vee p_2$ matches every string that matches p_1 or p_2, and $p_1 \& p_2$ is a pattern that matches every string matched by p_1 followed by any string that is matched by p_2. We have to be more precise here, that is we must describe in more detail how the *scanner* (the matching algorithm) works.

The patterns defined up till now consist of elementary subpatterns (strings *h*, *nil* and *fail*) connected by &- and $\vee$-operators. In a pattern of the form $p_1 \& p_2$, p_2 is called the *subsequent* of p_1, and in $p_1 \vee p_2$, p_2 is called the *alternative* of p_1. The scanner acts in the following way: if it has to match against a (sub-)pattern of the form $p_1 \vee p_2$, then the scanner behaves first as if it has encountered only the pattern p_1. But the alternative p_2, together with the current cursor position, will be remembered in case the match will fail later on.

If the matching process fails at a certain instant then something takes place which is called *backtracking*. The scanner returns to the last "choice point" in the pattern which is the last encountered subpattern of the form $p_1 \vee p_2$, where p_2 has not yet been tried. It restores the situation (cursor position) to how it was just before choosing p_1 as first alternative. It now chooses p_2 and the matching process proceeds as if p_2 had been substituted for $p_1 \vee p_2$ in the pattern.

If the scanner hits upon a pattern of the form $p_1 \& p_2$ then it first tries to match against p_1. If this fails then the backtracking process as described above will take place. If it succeeds then the scanner tries to match the subject against p_2 starting with a new cursor value which is the result of the match against p_1. If the scanner fails and all alternatives are exhausted which means that no more backtracking is possible, then the whole match fails. The overall match succeeds if we come to the right-hand side of the pattern.

EXAMPLE. We try to match the string 'arie' against the pattern

('a' & ('ri' $\vee$ 'r')) & 'i'.

1. Cursor position = 0. Match against 'a' succeeds. The new cursor position

is 1 and we try to match against the subsequent ('ri' v 'r') & 'i'.

2. This pattern has the form $p_1 \& p_2$, so we first try to match against the first component 'ri' v 'r'.
3. This pattern has the form $p_1 \vee p_2$. We now act as if we have encountered the pattern 'ri' alone. We remember however the situation as it is now in order to be able to backtrack to it later on.
4. Match against 'ri' succeeds, and the cursor value is now 3. We next try to match against the subsequent, the second component from step 2.
5. This is the pattern 'i'. Match against 'i' fails and thus we have to backtrack.
6. The situation of step 3 is restored by setting the cursor position to 1 again and we try the alternative 'r' instead of 'ri'.
7. Match against 'r' succeeds. The new cursor position is 2. We try the subsequent, the second component of ('ri' v 'r') & 'i'.
8. Match against this pattern 'i' succeeds, the new cursor position is 3.
9. There are no subsequents left and the whole match has thus succeeded.

Now that we know how the scanner works, we can describe the effect of the pattern *abort*. If the scanner encounters this pattern the whole matching process terminates. Notice the difference with *fail* which would force the scanner to backtrack.

We next discuss the way variables can be handled in the matching process. The variables will have strings as values. A variable *v* can be a pattern by itself. Such a pattern has the same meaning as the pattern *h*, where *h* is the value of *v* at the time the pattern expression is evaluated, which in general will be just before the match starts. There is an exception to this which will be discussed later on.

There are two ways to change values of variables during the match, namely by immediate and conditional assignment. Immediate assignment is indicated by patterns of the form *p$v*. The meaning of such a pattern is the following. If the scanner manages to match *p* against a substring of the subject, then this substring will be assigned to the variable *v*. This assignment is performed immediately, and always, even if the match will fail later on. Therefore a subpattern *p$v* can cause the variable *v* to be changed more than once in one match. For instance, during the match of the

the string 'arie' against the pattern (('a' ∨ 'ar') $ *v*) & 'ie' the variable *v* will change value twice. First the string 'a' will be assigned to it and later the string 'ar'.

Conditional assignment is indicated by patterns of the form *p.v*, and has a similar effect on *v* as immediate assignment, apart from the fact that the assignment will be performed only if the local match against *p* was part of a full match which led to success. Only after the match has terminated successfully, the corresponding assignments will be performed.

EXAMPLES. If 'arie' is matched against ('a'.*v* ∨ 'ar') & 'ie' then *v* will not get another value; if we match the same string against ('a' ∨ 'ar'.*v*) & 'ie' then after the match *v* will have the value 'ar'.

As we remarked above, evaluation of a pattern expression, which is in essence replacing variables by their values, will in general take place before the match. We now give the exception hinted at earlier, which is the *unevaluated expression *p*. This pattern behaves in the same way as *p* does except for the fact that *p* will be evaluated at the moment the scanner encounters **p* during the match (which therefore can happen more than once).

EXAMPLE. The pattern ('a'$*v* ∨ 'b'$*v*) & (**v*) matches the strings 'aa' and 'bb'. This can be contrasted with the pattern ('a'$*v* ∨ 'b'$*v*) & *v* which matches 'a' followed by *h* or 'b' followed by *h*, where *h* is the value of *v* before the match. Notice also that this latter pattern has the same effect as ('a'.*v* ∨ 'b'.*v*) & *v* or ('a'.*v* ∨ 'b'.*v*) & (**v*).

In the first example ('a'$*v* ∨ 'b'$*v*) & (**v*) we see a typical example of the use of the *-operator. We combined it there with the $-operator and were thus able to use the outcome of the match against the first component of the pattern while matching against the second component.

This concludes our informal discussion of the meaning of the patterns. The sequel of this paper will be devoted to more formal definitions of the process described above.

2. SYNTAX

We will now describe the syntax of the SNOBOL fragment, namely the pattern expressions that we allow in our language. We introduce the following syntactic classes, together with letters that denote typical elements of these classes.

$h \in \mathit{Str}$, the *strings*. String values will be enclosed in quotation marks, for instance 'arie' denotes the string consisting of letters a, r, i and e consecutively. We denote the empty string by ''. If $h \in \mathit{Str}$ and n,n' are integers such that $0 \le n \le n' \le k$, where k is the length of h, then $h[n:n']$ denotes the substring of h which begins with the $(n+1)$-th symbol and ends right after the n'-th symbol. If $0 \le n \le n' \le k$ does not hold then $h[n:n']$ denotes the empty string.

$v \in \mathit{Var}$, the *variables*. In contrast to full SNOBOL4, we define explicitly a class of variables which is distinct from the class of strings.

$n \in \mathit{Num}$, the *numerals* denoting nonnegative integers. We will use the letter n also to denote nonnegative integers themselves. No confusion will arise from this as the intended meaning can always be deduced from the context. We assume the existence of a *derepresentation function* $\mathcal{D}\colon \mathit{Num} \to \mathbb{N}$, mapping numerals to the corresponding nonnegative integers, which will be injective and surjective. This means that every integer has exactly one numeral representing it and therefore $\mathcal{D}^{-1}$ is well defined. This heavy machinery might seem somewhat overdone, and in fact we could, for the moment, do without it and proceed a little less formally. However we maintain this function here because in a later stage it is needed anyway, and also other $\mathcal{D}$-functions have to be introduced (see chapter 5, definition 5.1).

$p \in \mathit{Pat}$, the *patterns*. We give the following BNF-like definition.

p ::= $h\,\vert$	literal string
$v\,\vert$	variable
$\underline{nil}\,\vert\,\underline{abort}\,\vert\,\underline{fail}$	constants
$p\$v\,\vert$	immediate assignment
$p.v\,\vert$	conditional assignment

$n\$\$v \mid n..v \mid$	auxiliary patterns, not occurring in programs; needed in the operational semantics to describe the effect of immediate and conditional assignment
$*p \mid$	unevaluated expression
$p_1 \vee p_2 \mid$	alternation; we donot use the $\mid$-sign because the BNF notation does not allow this
$p_1 \& p_2$	concatenation; we chose the &-symbol instead of the space for the sake of clarity.

As we intend to study the matching process only we do not present the many other SNOBOL4 features. We also made a selection from the pattern structures which are possible in SNOBOL4. We have chosen the subset such that the essential aspects of the pattern matching process can be studied through it.

3. OPERATIONAL SEMANTICS

The semantics given here is inspired by a description of a SNOBOL4 implementation by GIMPEL [3]. There are some differences however.

The first one is that we allow only strings as values of variables. This has been done in order not to be forced to get into detail concerning coercion problems. Furthermore we do not include patterns as values because that would complicate the presentation a great deal, as we have to resort to recursively defined domains in the denotational semantics. This will be the subject of another paper.

Gimpel describes three phases in the elaboration of patterns. He talks about *compilation* which transforms a pattern expression into a tree representing it, *pattern building* which takes this tree, replaces variables by their values at that moment and builds a *pattern structure* (a graph representing the pattern tree in such a way that the scanner can traverse it efficiently), and *pattern matching*.

We do not distinguish these phases. Pattern match will be done directly from the pattern expression. We do not replace variables by their values but we add to the patterns a store *s* giving the values of the variables at pattern building time. In that way the scanner will be able, during the match, to find the meaning of a pattern component *v* by inspecting *s*.

The way the scanner performs the matching process, as described by Gimpel, is roughly as follows. Besides the cursor variable the scanner uses also the pattern structure which has been constructed by the pattern building process, and a variable, which we call *ptn* which has as a value a pointer into this structure indicating how far the match has proceeded in this structure. Furthermore there is a stack to save untried alternatives. The scanner now repeats the following loop.

1. If the pattern component pointed at by *ptn* has an alternative then save this alternative, represented by its *ptn* value, and the current cursor value on the stack.
2. Try to match the pattern component determined by the value of *ptn* (which is a primitive pattern: a string *h*, <u>*nil*</u>, <u>*fail*</u>, <u>*abort*</u>) against the subject string. If this succeeds goto step 3, else goto step 4.
3. Find out whether the pattern component designated by *ptn* has a subsequent. If so, set *ptn* to point at it and go back to step 1; if not, we are ready, and the overall match has succeeded.
4. (Backtrack step). Inspect the stack. If it is empty then we are done, there are no alternatives left, and the pattern match has failed. If the stack is not empty then pop the stack to find a new *ptn* value and a new cursor value. Assign these values and go back to step 1.

For those who are acquainted with SNOBOL4: the above algorithm describes the anchored fullscan mode, which will be the only mode to be dealt with in this paper.

Here the matching process will be defined in terms of an abstract machine, not unlike LANDIN's SECD machine [4]. We will give a function called *step* which performs the elementary steps of the process, changing the machine configuration, which we will now describe informally.

A machine configuration *m* is an 8-tuple $<p,s',h,c,a,q,n,s>$, where *p* is a pattern and *s'*,*s* are *stores* (lists of variable-value pairs). The store *s'* records the values of the variables as they were at pattern building time. The pair $<p,s'>$ determines a pattern structure, which corresponds roughly to the pattern component pointed at by the variable *ptn* in Gimpel's algorithm. Furthermore, h is the subject string, and c is the so called *subsequent* which is in essence a list of pattern components to be matched against once the match against p has terminated successfully. To be more precise, this is organized as follows: c is either equal to the list <READY>, the endmarker of the list, or it has the form $<p,s,h,c>$ where the pair $<p,s>$ determines the first pattern-structure component in the list, while c constitutes the tail of the list (the subject string h is included only for convenience). The item q in the 8-tuple is a list of variable-value pairs recording the conditional assignments encountered so far which have to be performed if the total match succeeds. Furthermore n is the numeral giving the present cursor value and s is the present store. Finally, the component a is the *alternative* which corresponds to the stack in Gimpel's description. This a is either equal to <FAIL> which denotes the bottom of the stack, or it is a 7-tuple of the form $<p,s,h,c,a,q,n>$. Here p, s and c correspond to the *ptn* value on the stack as given by Gimpel (p, s and c represent a point in the pattern structure: p and s determine a pattern structure component and c determines the subsequents of this component), n corresponds to the cursor value on Gimpel's stack, q denotes the queue of conditional assignments accumulated up to the moment that particular stack frame was constructed (this has no analogon in Gimpel's stack because he uses a trick to circumvent this space consuming method), and finally a stands for the other frames of the stack.

The function *step* takes a machine configuration $m=<p,s',h,c,a,q,n,s>$, inspects the form of p and changes m correspondingly into a new configuration. We next present the formal definitions.

First some notational conventions. We will frequently use Curried functions which are functions that can take functions as arguments and yield functions as values. This generally leads to expressions with too many parentheses to be readable. To avoid this we leave parentheses out as much as possible using the convention that function application associates to the left. This means that *fabc* should be taken as $((f(a))(b))(c)$.

Function domain parentheses will be omitted under the convention that the $\rightarrow$- operator associates to the right. That is, $A \rightarrow B \rightarrow C$ should be read as $A \rightarrow (B \rightarrow C)$.

Concatenation of two lists l_1 and l_2 will be written as $l_1{}^\wedge l_2$. The empty list will be denoted by <>.

We define the following classes.

$s \in S$, the *stores*. These are finite, and possibly empty, lists of elements from $Var \times Str$ ($<v,h>$-pairs). Elements from S are data structures which determine the values of the variables. Only the variables which have nonempty strings as values are recorded in a store s. In accordance with the convention in SNOBOL4 that all variables are initialized on the empty string there will be at any moment during program execution only a finite number of variables with nonempty values. We define two operations on stores.

1. updating. The store resulting from s by assigning the string h to v is represented by the list $s^\wedge <v,h>$.
2. extracting. The value of v in store s is denoted by $s(v)$. This is defined as follows.

a) $<>(v) = ''$

b) $(s^\wedge<v,h>)(w) = \begin{cases} h & \text{if } v \equiv w, \\ s(w) & \text{otherwise.} \end{cases}$

Notice that more than one pair with first element v can occur in a store s. Only the rightmost pair "counts" however.

$q \in Q$, the *queues of accumulated conditional assignments*. These too are finite and possibly empty lists of $<v,h>$-pairs. They constitute the queue of conditional assignments which have to be performed if the overall match succeeds. To accomplish this we simply concatenate the two lists: the store s' resulting from performing the assignments given by q in store s is given by $s' = s^\wedge q$

$c \in C$, the *subsequents*. The class C is inductively defined as follows. An element c from C is either the list <READY> containing one element, or it is a list of the form $<p,s,h,c>$ where $p \in Pat$, $h \in Str$, $s \in S$ and c is again a subsequent.

$a \in$ A, the *alternatives*. The class A is inductively defined by: an element a from A is either the one element list <FAIL> or a list $c^\wedge\langle a,q,n\rangle$ formed by concatenating a subsequent with a list containing an alternative, a $q \in$ Q and a numeral n.

$r \in$ R, the *results*, i.e. the possible outcomes of a match. The class R is defined by R = ($\mathit{Num} \cup$ {FAIL,ABORT}) $\times$ S. A results r is thus a pair $\langle\bar{n},s\rangle$ where $\bar{n}$ denotes the final cursor position (if the match was successful) and s is the resulting store.

$m \in$ M, the *machine configurations*. A machine configuration is either a final configuration which is an element from R, or an 8-tuple $a^\wedge\langle s\rangle$ formed by concatenating an alternative with a list containing a store s as only element. The predicate $\mathit{final}(m)$ holds iff m is a final configuration.

We now have enough tools to define the step function.

The function *step*: M $\to$ M is defined as follows.

A. If m is a final configuration then $\mathit{step}(m) = m$.

B. If m has the form $\langle$READY$,a,q,n,s\rangle$ then $\mathit{step}(m) = \langle n,s^\wedge q\rangle$.

C. In all other cases m has the form $\langle p,s',h,c,a,q,n,s\rangle$, and the definition proceeds by induction on the structue of p.

1. $\mathit{step}\langle h',s',h,c,a,q,n,s\rangle = \begin{cases} c^\wedge\langle a,q,\mathcal{D}^{-1}n'\rangle & \text{if } h' \equiv h[\mathcal{D}n:n'] \\ a^\wedge\langle s\rangle & \text{otherwise} \end{cases}$
2. $\mathit{step}\langle v,s',h,c,a,q,n,s\rangle = \langle s'(v),s',h,c,a,q,n,s\rangle$
3. $\mathit{step}\langle \underline{nil},s',h,c,a,q,n,s\rangle = c^\wedge\langle a,q,n,s\rangle$
4. $\mathit{step}\langle \underline{abort},s',h,c,a,q,n,s\rangle = \langle \text{ABORT},s\rangle$
5. $\mathit{step}\langle \underline{fail},s',h,c,a,q,n,s\rangle = a^\wedge\langle s\rangle$
6. $\mathit{step}\langle p\$v,s',h,c,a,q,n,s\rangle = \langle p,s',h,\langle n\$\$v,s',h,c\rangle,a,q,n,s\rangle$
7. $\mathit{step}\langle p.v,s',h,c,a,q,n,s\rangle = \langle p,s',h,\langle n..v,s',h,c\rangle,a,q,n,s\rangle$
8. $\mathit{step}\langle n'\$\$v,s',h,c,a,q,n,s\rangle = c^\wedge\langle a,q,n,s^\wedge\langle v,h[\mathcal{D}n':\mathcal{D}n]\rangle\rangle$
9. $\mathit{step}\langle n'..v,s',h,c,a,q,n,s\rangle = c^\wedge\langle a,q^\wedge\langle v,h[\mathcal{D}n':\mathcal{D}n']\rangle,n,s\rangle$
10. $\mathit{step}\langle *p,s',h,c,a,q,n,s\rangle = \langle p,s,h,c,a,q,n,s\rangle$
11. $\mathit{step}\langle p_1 \vee p_2,s',h,c,a,q,n,s\rangle = \langle p_1,s',h,c,\langle p_2,s',h,c,a,q,n\rangle,q,n,s\rangle$
12. $\mathit{step}\langle p_1 \& p_2,s',h,c,a,q,n,s\rangle = \langle p_1,s',h,\langle p_2,s',h,c\rangle,a,q,n,s\rangle$.

EXPLANATION.

Ad B. If during the match we encounter the end of the subsequent list, then the match has clearly succeeded (compare step 3 in Gimpel's algorithm). The postcursor position then is n, the cursor position on encountering READY, and the final store is obtained by performing all assignments in q from left to right.

Ad C. 1. If the pattern component is a literal string h' then we have to find out whether h' matches the subject string with respect to the present cursor position. If so, we have to continue with the next pattern component and this is given by the subsequent c. Now the store s and the queue q have not changed. Also there are no new alternatives found in the meantime so the alternative is still given by a. The only entity which has changed is the cursor position which must be set to its new value. By adding the list $<a,q,\mathcal{D}^{-1}n',s>$ to c we thus obtain a new machine configuration which reflects the effects of the successful match.

If the match against h' fails then we have to backtrack, and we take one frame from the stack a. Finding out that the match fails does not affect the store, so we only have to add $<s>$ to get the resulting machine configuration.

2. If the pattern component is a variable v then we have to inspect the store as it was at pattern building time to find the value of v. This store is given by s'. The resulting machine configuration is then obtained by replacing v by the literal string which is the value of v in s'.

6-8. The pattern $p\$v$ is handled as follows: $p\$v$ is rewritten as $p\,\&\,(n\$\$v)$. So first a match against p is attempted. If this succeeds then we have to assign to v the substring of the subject which has been matched, and that is precisely the effect of matching against $n\$\$v$. This match always succeeds and has the side effect that the substring from the subject between cursor value n (given by the pattern component $n\$\$v$) and the present cursor value is assigned to v. So $n\$\$v$ serves to indicate that an assignment has to be done, and it also provides the cursor value at the beginning of the match against p.

7-9. Similar to 6-8, but now the matched substring of the subject is added to the queue q.

10. The effect of matching against $*p$ in store s is the same as matching against the pattern structure derived from p in store s. So the only thing to be done is to replace s' by s.

11-12. In these cases the pattern is decomposed and the second component is retained in the new alternative, resp. the new subsequent.

Now that we have a step function which gives one step results, we can define the function $\mathcal{P}$ which takes a machine configuration and yields the final result of the matching process, a configuration m for which $final(m)$ holds. The function $\mathcal{P}$ is obtained by repeating the function *step* until a final configuration has been reached. We can formalize this in two ways. The first one is straightforward:

$$\mathcal{P}(m) = \begin{cases} m' & \text{if there exists a row } m_1,\dots,m_k \text{ such that } m = m_1, \\ & m_k = m',\ m_{i+1} = step(m_i)\ (i=1,\dots,k-1),\ \neg final(m_i) \\ & (i = 1,\dots,k-1),\ final(m_k), \\ \bot & \text{otherwise.} \end{cases}$$

There is another definition possible which is neater, but uses fixed point theory. The function $\mathcal{P}$ can be defined recursively by

$$\mathcal{P}(m) \Leftarrow final(m) \rightarrow m,\ \mathcal{P}(step(m)),$$

or more precisely

$$\mathcal{P} = \mu[\lambda\phi\cdot\lambda m\cdot final(m) \rightarrow m,\ \phi(step(m))],$$

where μ is the least fixed point operator (see for instance DE BAKKER [1], or STOY [7] who calls this operator *fix*).

In order for the latter definition to make sense we have to impose a cpo structure on the class M of machine configurations. This can be done by adding the element $\bot$ to M and making M a discrete cpo ($m_1 \sqsubseteq m_2$ iff $m_1 = \bot$ or $m_1 = m_2$). We also extend the definition of *step* by taking $step\ \bot = \bot$. It can be shown (in the standard way) that the operator $\lambda\phi\cdot\lambda m\cdot final(m) \rightarrow m,\ \phi(step(m))$ is continuous, and thus that the least fixed point exists.

That the two definitions are equivalent can be shown in a standard way (see for instance [1, paragraph 3.3]).

Finally we define the operational meaning function $\mathcal{O}$ which gives the outcome of the process of matching a string h against a pattern p with initial store s.

$$\mathcal{O}[\![p]\!]\ h\ s\ =\ \mathcal{P}\langle p,s,h,\langle\text{READY}\rangle,\langle\text{FAIL}\rangle,\langle\rangle,0,s\rangle.$$

Here we used the convention that syntactical objects occurring as an argument of a function are enclosed in $[\![\cdot]\!]$-type brackets to make the expression more readable.

4. DENOTATIONAL SEMANTICS

We now turn to a discussion of the denotational semantics of our SNOBOL4 fragment. Before we do so however, we first make a remark on the notation we will use. The semantical classes used in the denotational semantics will be different from the ones in the above chapter. For instance we defined the class S of stores by $S = (\mathit{Var} \times \mathit{Str})^*$, but now we will take the domain of the stores to be $S = \mathit{Var} \to \mathit{Str}$. This can be done because we do not work any more with finite representations, we can use infinitary mathematical objects such as functions in the denotational semantics.

We will however use the same symbols to denote corresponding semantical classes and their typical elements. So in this chapter we define S with typical element s by $s \in S = \mathit{Var} \to \mathit{Str}$. This usage will not cause confusion in this chapter because here we will be occupied only with denotational domains and values. If confusion can be possible we will use the so called *diacritical convention* (MILNE & STRACHEY [5]): elements in the denotational world will be decorated with an acute accent ´, and the operational domains and values with a grave accent `. According to this convention we then can write $\acute{S} = \mathit{Var} \to \mathit{Str}$ and $\grave{S} = (\mathit{Var} \times \mathit{Str})^*$. Notice that Var and Str, being syntactic domains are not decorated.

We return to the denotational semantics. The meaning of a pattern p can be described by the effects resulting from a match of an arbitrary string h against p. This match, if it succeeds, will affect the cursor value n (which is now an element of N, the nonnegative integers), the store s, and it might also add new conditional assignments to the ones already accumulated, as given by q (q will now be an element of $\acute{S} \to \acute{S}$ and denote the store transformation which is the result of performing all conditional assignments). The meaning of p will also be dependent on the store s' at pattern evaluation time, s' provides the values of the free variables in p, i.e. those variables that are not bound by a $*$-operator.

Using a meaning function N, the effect of matching the string h against pattern p evaluated in s', with initial situation given by conditional assignments q, cursor position n and store s, would then be given by the expression $N[\![p]\!]\ s'\ h\ q\ s$. The value of this expression could be a triple $<q',n',s''>$ giving the new q-, n- and s-values. This set up does not work however, and this can be seen most directly by studying the case that the match of h against p fails. For how should the effect of backtracking be described in this setting?

The problem becomes clearer if we take a look at the compositionality principle, a main idea behind the denotational style of defining. This principle says that the meaning of a compound expression should be composed from the meaning of its parts. For instance, the meaning $N[\![(p_1 \vee p_2)\ \&\ p_3]\!]$ should be given in terms of $N[\![p_1]\!]$, $N[\![p_2]\!]$ and $N[\![p_3]\!]$ only.

Now matching against p_3 can fail and cause backtracking, a jump in the pattern to p_2. However in determining the meaning $N[\![p_3]\!]$ we donot have the pattern text p_2 at our disposal anymore, as was the case in the operational semantics. The standard solution for this kind of problems around jumps in programs is to work with continuations.

The trick is that we give the function $N[\![p]\!]\ s'\ h$ an extra argument a, called the alternative which describes the result of backtracking from p. The effect of backtracking is: recover the situation to the state it was in at the latest choice point and proceed from there on *with the new store s*. This effect is captured by a function $a \in A = S \to R$, where $R = (N \cup \{FAIL, ABORT\}) \times S$. The alternative a takes a store as argument and delivers the result r of the rest of the matching process. In other words, an alternative a is a function that describes the pattern matching process starting from the moment that backtracking out of p occurs.

So we add an extra argument a, and we have that now, if backtracking takes place, $N[\![p]\!]\ s'\ h\ a\ q\ n\ s$ denotes the final result of the whole match.

But this must then be the case too should the match succeed. It is therefore needed to give $\mathcal{N}[\![p]\!]\ s'\ h$ yet another argument, a subsequent c, describing how the pattern matching process proceeds if matching against p has terminated successfully. The subsequent will yield a result r in R, which will be dependent on four arguments describing the situation after the local match has succeeded: the new store s'', the postcursor position n', the conditional assignments accumulated q' and a new alternative a' which is determined by the alternative we had before matching against p, updated with the possible alternatives found while matching against p which have not yet been tried. We thus arrive at a functionality $c \in C = A \to Q \to N \to S \to R$.

A subsequent can be viewed as a function determining how the match proceeds from a certain point in the pattern text. An alternative can be looked upon in the same way, but more information is available at the moment an alternative is constructed (i.e. while matching on encountering a choice point $p_1 \vee p_2$), namely the precursor position, the conditional assignments gathered so far and also the alternatives remaining if the match fails in the process after backtracking to the choice point. The difference between the two is clearly reflected in the respective functionalities $A \to Q \to N \to S \to R$ vs. $S \to R$: an alternative is like a subsequent but not more dependent on a, q and n.

Concluding, the result $r = \mathcal{N}[\![p]\!]\ s'\ h\ c\ a\ q\ n\ s$ can be described as follows: $\mathcal{N}[\![p]\!]\ s'$ denotes the pattern structure resulting from evaluation of the expression p in store s'. Suppose h is matched against this pattern structure, and the initial situation is given by cursor position n, initial store s and conditional assignments accumulated so far determined by q. Suppose furthermore that the effect of the future of the matching process once match against p has been finished is given by a in case backtracking out of p occurs, and by c in case the match against p terminates successfully. In that case the final result of the whole matcing process is given by r.

The above discussion leads to the following definitions of domains and functionalities.

$s \in S = \mathit{Var} \to \mathit{Str}$	stores
$n \in N$	nonnegative integers
$q \in Q = S \to S$	accumulated conditional assignments
$r \in R = (N \cup \{FAIL, ABORT\}) \times S$	results
$a \in A = S \to R$	alternatives
$c \in C = A \to Q \to N \to S \to R$	subsequents

We define a *variant* $s\{h/v\}$ of a store s by

$$(s\{h/v\})(w) = \begin{cases} s(w) & \text{if } v \not\equiv w \\ h & \text{if } v \equiv w. \end{cases}$$

Notice that the classes introduced above are not cpo's. Cpo's are not needed here because the semantic definition of $\mathcal{N}$ to come is a purely inductive one. No use is made of fixed points, and we also donot use recursively defined domains.

The semantic function $\mathcal{N}$ has functionality

$$\mathcal{N}\colon \mathit{Pat} \to S \to \mathit{Str} \to C \to A \to Q \to N \to S \to R$$

and is defined by induction on the structure of its first argument as follows.

1. $\mathcal{N}\llbracket h' \rrbracket\ s'\ h\ c\ a\ q\ n\ s = \begin{cases} c\ a\ q\ n'\ s & \text{if } h' \equiv h[n{:}n'] \\ a\ s & \text{otherwise} \end{cases}$
2. $\mathcal{N}\llbracket v \rrbracket\ s'\ h\ c\ a\ q\ n\ s = \mathcal{N}\llbracket s'(v) \rrbracket\ s'\ h\ c\ a\ q\ n\ s.$
3. $\mathcal{N}\llbracket \underline{nil} \rrbracket\ s'\ h\ c\ a\ q\ n\ s = c\ a\ q\ n\ s$
4. $\mathcal{N}\llbracket \underline{abort} \rrbracket\ s'\ h\ c\ a\ q\ n\ s = \langle ABORT, s\rangle$
5. $\mathcal{N}\llbracket \underline{fail} \rrbracket\ s'\ h\ c\ a\ q\ n\ s = a\ s$
6. $\mathcal{N}\llbracket p\$v \rrbracket\ s'\ h\ c\ a\ q\ n\ s = \mathcal{N}\llbracket p \rrbracket\ s'\ h\ c'\ a\ q\ n\ s$
   where $c' = \lambda a'\cdot\lambda q'\cdot\lambda n'\cdot\lambda s''\cdot c\ \ a'\ q'\ n'(s''\{h[n{:}n']/v\})$
7. $\mathcal{N}\llbracket p\cdot v \rrbracket\ s'\ h\ c\ a\ q\ n\ s = \mathcal{N}\llbracket p \rrbracket\ s'\ h\ c'\ a\ q\ n\ s$
   where $c' = \lambda a'\cdot\lambda q'\cdot\lambda n'\cdot\lambda s''\cdot c\ a'(\lambda s\cdot(q's)\{h[n{:}n']/v\})n's''$
8. $\mathcal{N}\llbracket \bar{n}\$\$v \rrbracket\ s'\ h\ c\ a\ q\ n\ s = c\ a\ q\ n(s\{h[\mathcal{D}\bar{n}{:}n]/v\})$
9. $\mathcal{N}\llbracket \bar{n}..v \rrbracket\ s'\ h\ c\ a\ q\ n\ s = c\ a(\lambda s\cdot(q\ s)\{h[\mathcal{D}\bar{n}{:}n]/v\})\ n\ s$
10. $\mathcal{N}\llbracket *p \rrbracket\ s'\ h\ c\ a\ q\ n\ s = \mathcal{N}\llbracket p \rrbracket\ s\ h\ c\ a\ q\ n\ s$
11. $\mathcal{N}\llbracket p_1 \vee p_2 \rrbracket\ s'\ h\ c\ a\ q\ n\ s = \mathcal{N}\llbracket p_1 \rrbracket\ s'\ h\ c\{\mathcal{N}\llbracket p_2 \rrbracket\ s'\ h\ c\ a\ q\ n\}\ q\ n\ s$
12. $\mathcal{N}\llbracket p_1 \& p_2 \rrbracket\ s'\ h\ c\ a\ q\ n\ s = \mathcal{N}\llbracket p_1 \rrbracket\ s'\ h\{\mathcal{N}\llbracket p_2 \rrbracket\ s'\ h\ c\}\ a\ q\ n\ s.$

REMARKS. Ad 1. If h matches h' at cursor position n then the remainder of the matching process is given by the subsequent c. The alternative, the conditional assignment queue and the store did not change, only the cursor has a new value. If h does not match then the remainder of the matching process is given by a which has to be applied to the current store s.

Ad 2. Like 1, but first the value of v in s' has to be determined. Notice that, in order to be able to maintain the above definition as one by structural induction, we have to choose a measure of complexity of patterns which guarantees that the complexity of variable is higher than that of a string. This can be accomplished easily though.

Ad 6. Matching gainst $p\$v$ is in principle the same as matching against p. Only when the match against p succeeds we have to aditionally assign the string matched to. This is taken care of by the new subsequent c' which describes the effect of this assignment followed by the effect of the old subsequent c.

Ad 7. Like 6, but now the new subsequent c' causes q to be updated instead of s.

Ad 8,9. Notice that the $\bar{n}$ occurring in the patterns is a numeral. Therefore $\bar{n}$ has to be changed into the corresponding number. Strictly speaking, clauses 8 and 9 are not needed in the definition, because patterns $\bar{n}\$\$v$, $\bar{n}..v$ do not occur in programs, nor in the right-hand sides of the other clauses in the definition. These auxiliary patterns have only been introduced for the sake of the operational definition, where the meaning of $p\$v$ has been defined in terms of the meaning of some $\bar{n}\$\$v$. See also the lemmas at the end of this chapter. We maintained these clauses here, because we will need them in proving the operational and denotational semantics equivalent.

Ad 11. Matching against $p_1 \vee p_2$ amounts to matching against p_1, but now we have a new alternative. On backtracking we have to match against p_2 in the situation as it is now (apart from the new store). This effect is taken care of by the new alternative $N[\![p_2]\!]\ s'\ h\ c\ a\ q\ n$. Notice that this alternative has the right functionality.

Ad 12. As in 11, but now a new subsequent is formed.

Finally we define the denotational counterpart of the function $\mathcal{O}$ from chapter 3. This is the semantic function $\mathcal{M}$ with functionality $\mathcal{M}$: $\mathit{Pat} \to \mathit{Str} \to \mathrm{S} \to \mathrm{R}$.

$$\mathcal{M}[\![p]\!]\ h\ s = \mathcal{N}[\![p]\!]\ s\ \ h\ \ \mathit{ready}\ \mathit{fail}\ \{\lambda s\cdot s\}\ 0\ s,$$

where the subsequent *ready* is defined by

$$\mathit{ready}\ a\ q\ n\ s = \langle n, q\ s\rangle$$

and the alternative *fail* by

$$\mathit{fail}\ s = \langle \mathrm{FAIL}, s\rangle.$$

So the complete matching of a string h against a pattern p in store s, corresponds to evaluating p in s, and matching h against this pattern structure. If this match succeeds then the accumulated conditional assignments have to be performed and this is handled by the subsequent *ready* which yields the postcursor position and the updated store. If the match fails then this has to be reported and that is what the alternative *fail* is for. Furthermore, the precursor position is 0, and the initial store is s. Finally, in the beginning there are no conditional assignments accumulated and this is denoted by the identity function $\lambda s\cdot s$.

We close this chapter by giving a lemma on the relation between clauses 6 and 8 (7 and 9) of the definition of $\mathcal{N}$.

LEMMA 4.1.
1. $\mathcal{N}[\![p\$v]\!]\ s'\ h\ c\ a\ q\ n\ s = \mathcal{N}[\![p\&(\bar{n}\$\$v)]\!]\ s'\ h\ c\ a\ q\ n\ s$
2. $\mathcal{N}[\![p.v]\!]\ s'\ h\ c\ a\ q\ n\ s = \mathcal{N}[\![p\&(\bar{n}..v)]\!]\ s'\ h\ c\ a\ q\ n\ s$

where $\bar{n} = \mathcal{D}^{-1}n$.

PROOF. The proof is straightforward by writing out the expressions. For instance in 1 we have: left-hand side = $\mathcal{N}[\![p]\!]\ s'\ h\ c'\ a\ q\ n\ s$, where $c'\ a'\ q'\ n'\ s'' = c\ a'\ q'\ n'(s''\{h[n{:}n']/v\})$, and right-hand side = $\mathcal{N}[\![p]\!]\ s'\ h\{\mathcal{N}[\![\bar{n}\$\$v]\!]s'\ h\ c\}\ a\ q\ n\ s$. So there remains to be proved $c' = \mathcal{N}[\![\bar{n}\$\$v]\!]\ s'\ h\ c$, and this follows from the fact that $\mathcal{N}[\![\bar{n}\$\$v]\!]\ s'\ \ h\ c\ a'\ q'\ n'\ s'' = c'\ a'\ q'\ n'(s''\{h[\mathcal{D}\bar{n}{:}n']/v\})$ and that $\mathcal{D}\bar{n} = \mathcal{D}\mathcal{D}^{-1}n = n$. □

5. OPERATIONAL AND DENOTATIONAL SEMANTICS ARE EQUIVALENT

Of course it is not by coincidence that the two semantics presented here are that similar. An example of the difficulties that one encounters if one chooses a more dissimilar pair of semantics will be given in the next chapter. Notice however, that there are essential differences between the two semantics. A first one is that the objects that are manipulated in the operational semantics are all finite representations (they are in fact BNF-definable) while the denotational semantics handles infinitary abstract objects. A more fundamental difference is that the denotational semantics is fully compositional while the operational semantics is not. Related to this is the fact that in the denotational semantics the outcome of the matching process is obtained, so to speak immediately, by applying the meaning function $\mathcal{M}$ to the suitable arguments, while in the operational semantics we get the result by letting an abstract machine compute it step by step.

If now we want co compare the two semantics the first thing to do is to find a correspondence between the operational domains and the denotational ones. The main theorem to be proved here is that the two functions $\mathcal{O}$ and $\mathcal{M}$, applied to corresponding arguments, will yield corresponding results. There is a straightforward correspondence between the domains, which will be given by the *derepresentation functions* $\mathcal{D}_X$ (one for every pair of domains $\acute{X}$ and $\check{X}$) which map an element from the operational domain $\check{X}$ onto the corresponding element in $\acute{X}$. So we will define functions $\mathcal{D}_S$, $\mathcal{D}_N$ (this is the function introduced already in chapter 2, which relates numerals and numbers), $\mathcal{D}_Q$, $\mathcal{D}_R$, $\mathcal{D}_A$ and $\mathcal{D}_C$. In the sequel we will use the convention that the subscripts will be omitted if this causes no confusion (this has already been done in chapter 2). We remark now already that these functions $\mathcal{D}$ will in general be neither one to one nor onto.

The fact to be proved in this chapter can now be stated as $\mathcal{D}(\mathcal{O}[\![p]\!]\ h\ s) = \mathcal{M}[\![p]\!]\ h\ (\mathcal{D}s)$. We will first give the definitions and afterwards provide some comments on these.

DEFINITION 5.1.

$\mathcal{D}_S\colon \grave{S} \to \acute{S}$ is defined by $\mathcal{D}_S\langle\rangle = \lambda v\cdot''$

$$\mathcal{D}_S(\grave{s}^\wedge\langle v,h\rangle) = (\mathcal{D}_S\grave{s})\{h/v\}$$

$\mathcal{D}_Q\colon \grave{Q} \to \acute{Q}$ is defined by $\mathcal{D}_Q\langle\rangle = \lambda\acute{s}\cdot\acute{s}$

$$\mathcal{D}_Q(\grave{q}^\wedge\langle v,h\rangle) = \lambda\acute{s}\cdot((\mathcal{D}_Q\grave{q}\acute{s})\{h/v\})$$

$\mathcal{D}_R\colon \grave{R} \to \acute{R}$ is defined by $\mathcal{D}_R\langle\bar{n},\grave{s}\rangle = \langle\mathcal{D}_{\bar{N}}\bar{n},\mathcal{D}_S\grave{s}\rangle$

where $\mathcal{D}_{\bar{N}}\colon \mathit{Num} \cup \{\text{FAIL},\text{ABORT}\} \to N \cup \{\text{FAIL},\text{ABORT}\}$

is defined by $\begin{cases} \mathcal{D}_N\bar{n} & \text{if } \bar{n} \in \mathit{Num} \\ \bar{n} & \text{if } \bar{n} \in \{\text{FAIL},\text{ABORT}\} \end{cases}$

$\mathcal{D}_C\colon \grave{C} \to \acute{C}$ is defined by $\mathcal{D}_C\langle\text{READY}\rangle = \mathit{ready}$

$$\mathcal{D}_C\langle p,\grave{s},h,\grave{c}\rangle = \mathcal{N}[\![p]\!]\,(\mathcal{D}_S\grave{s})\;h\;(\mathcal{D}_C\grave{c})$$

$\mathcal{D}_A\colon \grave{A} \to \acute{A}$ is defined by $\mathcal{D}_A\langle\text{FAIL}\rangle = \mathit{fail}$

$$\mathcal{D}_A(\grave{c}^\wedge\langle\grave{a},\grave{q},\grave{n}\rangle) = (\mathcal{D}_C\grave{c})\,(\mathcal{D}_A\grave{a})\,(\mathcal{D}_Q\grave{q})\,(\mathcal{D}_N\grave{n}).$$

REMARKS. We have $\mathcal{D}_S\colon (\mathit{Var} \times \mathit{Str})^* \to (\mathit{Var} \to \mathit{Str})$. Now the empty list corresponds to the situation that all variables have the empty string as value, so this accounts for the first line in the definition. Furthermore in the store $\grave{s}^\wedge\langle v,h\rangle$ all variables have the same value as in $\grave{s}$, except for v which has the value h, and this is reflected in the second line of the definition.

The function $\mathcal{D}_N$ has already been introduced in the second chapter. It has not been defined there, and we could not do so because we chose not to define the form of the elements in Num.

The functionality of $\mathcal{D}_Q$ is $(\mathit{Var} \times \mathit{Str})^* \to (\acute{S} \to \acute{S})$. The queue $\grave{q}$ in $\grave{Q}$ provides the conditional assignments to be performed from left to right. The corresponding function $\mathcal{D}_Q\grave{q}$ is the store transformation that describes the effect of performing these assignments. So we have $\mathcal{D}_Q\langle\rangle = \lambda\acute{s}\cdot\acute{s}$, for if the queue is empty then the store does not change. The second line of the definition can be phrased as follows: performing the assignments in the queue $\grave{q}^\wedge\langle v,h\rangle$ amounts to performing the assignments in $\grave{q}$ first and afterwards assigning h to v.

The function $\mathcal{D}_R$ is defined straightforwardly.

That $\mathcal{D}_C$<READY> should be equal to *ready* can be seen from the fact that they "do the same job": the accumulated conditional assignments are performed and the final result is delivered. This is formalized in Lemma 5.3. The next line in the definition can be commented upon as follows. The subsequent $<p,\tilde{s},h,\tilde{c}>$ describes a match of h against p evaluated in $\tilde{s}$, followed by subsequent $\tilde{c}$, and the entity $\mathcal{N}[\![p]\!](\mathcal{D}_S\tilde{s})\ h\ (\mathcal{D}_C\tilde{c})$ describes the same process for the corresponding elements in the denotational world.

Similar remarks as given on $\mathcal{D}_C$ apply for the function $\mathcal{D}_A$.

We next state some lemmas giving results on these functions.

LEMMA 5.2. $\mathcal{D}(\tilde{s}^\wedge\tilde{q}) = (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{s})$.

PROOF. Remind that the list $\tilde{s}^\wedge\tilde{q}$ represents the store resulting from performing the conditional assignments in $\tilde{q}$ on $\tilde{s}$ (see the definition of *step*<READY,...> in chapter 3). The proof is by induction on the length of $\tilde{q}$.

Basis. $\mathcal{D}(\tilde{s}^\wedge<>) = \mathcal{D}\tilde{s}$ and $(\mathcal{D}<>)(\mathcal{D}\tilde{s}) = (\lambda\acute{s}\cdot\acute{s})(\mathcal{D}\tilde{s}) = \mathcal{D}\tilde{s}$.

Induction step. $\mathcal{D}(\tilde{s}^\wedge(\tilde{q}^\wedge<v,h>)) = \mathcal{D}((\tilde{s}^\wedge\tilde{q})^\wedge<v,h>) =$
$(\mathcal{D}(\tilde{s}^\wedge\tilde{q}))\{h/v\}$.

On the other hand $(\mathcal{D}(\tilde{q}^\wedge<v,h>))(\mathcal{D}\tilde{s}) = [\lambda\acute{s}\cdot((\mathcal{D}\tilde{q}\acute{s})\{h/v\})](\mathcal{D}\tilde{s}) =$
$[(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{s})]\{h/v\}$,

and the result holds by induction. □

LEMMA 5.3. $\mathcal{D}(step$<READY,$\tilde{a},\tilde{q},\tilde{n},\tilde{s}$>$) = ready(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$.

PROOF. The left-hand side is equal to $\mathcal{D}<\tilde{n},\tilde{s}^\wedge\tilde{q}> = <\mathcal{D}\tilde{n},\mathcal{D}(\tilde{s}^\wedge\tilde{q})>$, and the right-hand side equals $<\mathcal{D}\tilde{n},(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{s})>$. The result now holds by the preceding lemma. □

LEMMA 5.4. $\mathcal{D}(\mathcal{P}$<READY,$\tilde{a},\tilde{q},\tilde{n},\tilde{s}$>$) = ready(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$.

PROOF. Because *step*<READY,$\tilde{a},\tilde{q},\tilde{n},\tilde{s}$> is final, we have that $\mathcal{P}$<READY,$\tilde{a},\tilde{q},\tilde{n},\tilde{s}$> is equal to this, and the lemma immediately follows from Lemma 5.3. □

LEMMA 5.5. $\mathcal{D}$<FAIL,$\tilde{s}$> $= fail(\mathcal{D}\tilde{s})$.

PROOF. Immediate by writing out the expressions. □

LEMMA 5.6. $\tilde{s}\llparenthesis v \rrparenthesis = (\mathcal{D}\tilde{s})(v)$.

PROOF. Induction on the length of $\tilde{s}$. The basic step is OK because $<>\llparenthesis v \rrparenthesis = ''$ and $(\mathcal{D}<>)(v) = (\lambda v \cdot '')(v) = ''$.

Induction step.

$$(\tilde{s}^\wedge <w,h>)\llparenthesis v \rrparenthesis = \begin{cases} h & \text{if } w \equiv v \\ \tilde{s}\llparenthesis v \rrparenthesis & \text{otherwise} \end{cases}$$

$$\mathcal{D}(\tilde{s}^\wedge <w,h>)(v) = ((\mathcal{D}\tilde{s})\{h/w\})(v) = \begin{cases} h & \text{if } w \equiv v \\ (\mathcal{D}\tilde{s})(v) & \text{otherwise.} \end{cases} \qquad \square$$

Now we want to prove $\mathcal{D}(\mathcal{O}\llbracket p \rrbracket h\ \tilde{s}) = \mathcal{M}\llbracket p \rrbracket\ h\ (\mathcal{D}\tilde{s})$. By writing out, using the definition of $\mathcal{O}$ and $\mathcal{M}$, this is equivalent to

$$\mathcal{D}(\mathcal{P}<p,\tilde{s},h,<\text{READY}>,<\text{FAIL}>,<>,0,\tilde{s}>) =$$
$$\mathcal{N}\llbracket p \rrbracket (\mathcal{D}\tilde{s})\ h\ \mathit{ready}\ \mathit{fail}\ (\lambda \tilde{s} \cdot \tilde{s})\ 0\ (\mathcal{D}\tilde{s}).$$

We distinghuish two cases, namely that the left-hand side of the above equality is unequal to $\perp$ and the case that it is equal to $\perp$. We establish the desired result for the first case by proving the following more general result.

LEMMA 5.7. For all $\tilde{c},\tilde{a},\tilde{q},\tilde{n}$ and $\tilde{s}$ we have: if $\mathcal{P}(\tilde{c}^\wedge <\tilde{a},\tilde{q},\tilde{n},\tilde{s}>) \neq \perp$ then $\mathcal{D}(\mathcal{P}(\tilde{c}^\wedge <\tilde{a},\tilde{q},\tilde{n},\tilde{s}>)) = (\mathcal{D}\tilde{c})(\mathcal{D}\tilde{a})(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$.

PROOF. The proof is by induction, essentially on the length of the computation. Now we have given two definitions of $\mathcal{P}$, and the induction argument depends on the definition chosen. If one thinks in terms of the fixed point definition then we have to use Scott's induction (fixed point induction), that is we have to prove that the lemma holds for $\lambda m \cdot \perp$ instead of $\mathcal{P}$ (which is clearly true), and that the lemma holds for $\lambda m \cdot \mathit{final}(m) \rightarrow m,\ \Phi(\mathit{step}(m))$ given that the lemma holds for the function Φ instead of $\mathcal{P}$. The proof given below can be reorganized in these terms.

If one adopts the other definition using rows of machine configurations, then the induction is simply on the length of the row. The basic step is again vacuously fulfilled because one easily sees that there are no zero step reductions from $\check{c}^\wedge\langle\check{a},\check{q},\check{n},\check{s}\rangle$ since this configuration is not final. For the induction steps we distinguish thirteen cases, dependent on the form of $\check{c}$.

1. $\check{c} = \langle\text{READY}\rangle$. The lemma holds by Lemma 5.4.

2. $\check{c} = \langle h',\check{s}_1,h,\check{c}_1\rangle$. There are two cases:
   a. $h[\mathcal{D}\check{n}:\bar{n}] \equiv h'$ for some $\bar{n}$. We then have to prove
   $\mathcal{D}(P(\check{c}_1{}^\wedge\langle\check{a},\check{q},\mathcal{D}^{-1}\bar{n},\check{s}\rangle)) = (\mathcal{D}\check{c}_1)(\mathcal{D}\check{a})(\mathcal{D}\check{q})\ \bar{n}\ (\mathcal{D}\check{s})$
   and this holds by induction and the fact that $\mathcal{D}\ \mathcal{D}^{-1}\ \bar{n} = \bar{n}$.
   b. Otherwise. Then the property to be proven is equivalent to
   $\mathcal{D}(P(\check{a}^\wedge\langle\check{s}\rangle)) = (\mathcal{D}\check{a})(\mathcal{D}\check{s})$. Again there are two cases:
   I. $\check{a} = \langle\text{FAIL}\rangle$. In this case we have to prove
   $\mathcal{D}(P\langle\text{FAIL},\check{s}\rangle) = \mathit{fail}(\mathcal{D}\check{s})$ and this is an immediate consequence of Lemma 5.5.
   II. $\check{a} = \check{c}_2{}^\wedge\langle\check{a}_1,\check{q}_1,\check{n}_1\rangle$. Then we have to prove
   $\mathcal{D}(P\langle\check{c}_2,\check{a}_1,\check{q}_1,\check{n}_1,\check{s}\rangle) = (\mathcal{D}\check{c}_2)(\mathcal{D}\check{a}_1)(\mathcal{D}\check{q}_1)(\mathcal{D}\check{n}_1)(\mathcal{D}\check{s})$
   and this holds by induction.

3. $\check{c} = \langle v,\check{s}_1,h,\check{c}_1\rangle$. We have to prove
   $\mathcal{D}(P\langle\check{s}_1(\!|v|\!),\check{s}_1,h,\check{c}_1,\check{a},\check{q},\check{n},\check{s}\rangle) =$
   $\mathcal{N}[\![v]\!](\mathcal{D}\check{s}_1)h(\mathcal{D}\check{c}_1)(\mathcal{D}\check{a})(\mathcal{D}\check{q})(\mathcal{D}\check{n})(\mathcal{D}\check{s}) =$
   $\mathcal{N}[\![(\mathcal{D}\check{s}_1)(v)]\!](\mathcal{D}\check{s}_1)h(\mathcal{D}\check{c}_1)(\mathcal{D}\check{a})(\mathcal{D}\check{q})(\mathcal{D}\check{n})(\mathcal{D}\check{s}) = (\#)$
   We have by Lemma 5.6 and the definition of $\mathcal{D}_C$ that
   $(\#) = (\mathcal{D}\langle\check{s}_1(\!|v|\!),\check{s}_1,h,\check{c}_1\rangle)(\mathcal{D}\check{a})(\mathcal{D}\check{q})(\mathcal{D}\check{n})(\mathcal{D}\check{s})$,
   and now we can apply the induction hypothesis.

4. $\check{c} = \langle\underline{nil},\check{s}_1,h,\check{c}_1\rangle$. Like 2a.

5. $\check{c} = \langle\underline{abort},\check{s}_1,h,\check{c}_1\rangle$. Immediate.

6. $\check{c} = \langle\underline{fail},\check{s}_1,h,\check{c}_1\rangle$. Like 2b.

7. $\check{c} = \langle p\$v,\check{s}_1,h,\check{c}_1\rangle$. We have to prove
   $\mathcal{D}(P\langle p,\check{s}_1,h,\langle\check{n}\$\$v,\check{s}_1,h,\check{c}_1\rangle,\check{a},\check{q},\check{n},\check{s}\rangle) =$
   $\quad\mathcal{N}[\![p\$v]\!](\mathcal{D}\check{s}_1)h(\mathcal{D}\check{c}_1)(\mathcal{D}\check{a})(\mathcal{D}\check{q})(\mathcal{D}\check{n})(\mathcal{D}\check{s})$.
   We have
   $\mathcal{D}(P\langle p,\check{s}_1,h,\langle\check{n}\$\$v,\check{s}_1,h,\check{c}_1\rangle,\check{a},\check{q},\check{n},\check{s}\rangle) =$ (ind.hyp.)
   $(\mathcal{D}\langle p,\check{s}_1,h,\langle\check{n}\$\$v,\check{s}_1,h,\check{c}_1\rangle\rangle)(\mathcal{D}\check{a})(\mathcal{D}\check{q})(\mathcal{D}\check{n})(\mathcal{D}\check{s}) =$ (def. $\mathcal{D}_C$)
   $(\mathcal{N}[\![p]\!](\mathcal{D}\check{s}_1)h\{\mathcal{N}[\![\check{n}\$\$v]\!](\mathcal{D}\check{s}_1)h(\mathcal{D}\check{c}_1)\})(\mathcal{D}\check{a})(\mathcal{D}\check{q})(\mathcal{D}\check{n})(\mathcal{D}\check{s}) =$ (def. $\mathcal{N}$)
   $\mathcal{N}[\![p\&(\check{n}\$\$v)]\!](\mathcal{D}\check{s}_1)h(\mathcal{D}\check{c}_1)(\mathcal{D}\check{a})(\mathcal{D}\check{q})(\mathcal{D}\check{n})(\mathcal{D}\check{s}) =$

$N[\![p\$v]\!](\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s})$.
The last equality holds by Lemma 4.1 and the fact that $\mathcal{D}^{-1}\mathcal{D}\grave{n} = \grave{n}$.

8. $\grave{c} = <p.v,\grave{s}_1,h,\grave{c}_1>$. Analogous to 7.

9. $\grave{c} = <\grave{n}_1\$\$v,\grave{s}_1,h,\grave{c}_1>$. We have to prove
$\mathcal{D}(\mathcal{P}(\grave{c}_1{}^\wedge<\grave{a},\grave{q},\grave{n},\grave{s}^\wedge<v,h[\mathcal{D}\grave{n}_1:\mathcal{D}\grave{n}]>>)) =$
$(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})((\mathcal{D}\grave{s})\{h[\mathcal{D}\grave{n}_1:\mathcal{D}\grave{n}]/v\})$
and this holds by induction and the definition of $\mathcal{D}_S$.

10. $\grave{c} = <\grave{n}_1..v,\grave{s}_1,h,\grave{c}_1>$. We have to prove
$\mathcal{D}(\mathcal{P}(\grave{c}_1{}^\wedge<\grave{a},\grave{q}^\wedge<v,h[\mathcal{D}\grave{n}_1:\mathcal{D}\grave{n}]>,\grave{n},\grave{s}>)) =$
$(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\lambda\acute{s}\cdot(\mathcal{D}\grave{q}\acute{s})\{h[\mathcal{D}\grave{n}_1:\mathcal{D}\grave{n}]/v\})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s})$
and this holds by induction and the definition of $\mathcal{D}_Q$.

11. $\grave{c} = <*p,\grave{s}_1,h,\grave{c}_1>$. We have to prove
$\mathcal{D}(\mathcal{P}<p,\grave{s},h,\grave{c}_1,\grave{a},\grave{q},\grave{n},\grave{s}>) = N[\![p]\!](\mathcal{D}\grave{s})h(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s})$,
which holds by induction and the definition of $\mathcal{D}_C$.

12. $\grave{c} = <p_1 \vee p_2,\grave{s}_1,h,\grave{c}_1>$. We have
$\mathcal{D}(\mathcal{P}(\grave{c}^\wedge<\grave{a},\grave{q},\grave{n},\grave{s}>)) =$
$\mathcal{D}(\mathcal{P}<p_1,\grave{s}_1,h,\grave{c}_1,<p_2,\grave{s}_1,h,\grave{c}_1,\grave{a},\grave{q},\grave{n}>,\grave{q},\grave{n},\grave{s}>) =$ (ind. hyp.)
$(\mathcal{D}<p_1,\grave{s}_1,h,\grave{c}_1>)(\mathcal{D}<p_2,\grave{s}_1,h,\grave{c}_1,\grave{a},\grave{q},\grave{n}>)(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) =$ (def. $\mathcal{D}_C,\mathcal{D}_A$)
$N[\![p_1]\!](\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)\{N[\![p_2]\!](\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})\}(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) =$
$N[\![p_1 \vee p_2]\!](\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) =$ (def. $\mathcal{D}_C$)
$(\mathcal{D}<p_1 \vee p_2,\grave{s}_1,h,\grave{c}_1>)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s})$.

13. $\grave{c} = <p_1 \& p_2,\grave{s}_1,h,\grave{c}_1>$. We have
$\mathcal{D}(\mathcal{P}(\grave{c}^\wedge<\grave{a},\grave{q},\grave{n},\grave{s}>)) =$
$\mathcal{D}(\mathcal{P}<p_1,\grave{s}_1,h,<p_2,\grave{s}_1,h,\grave{c}_1>,\grave{a},\grave{q},\grave{n},\grave{s}>) =$ (ind. hyp.)
$(\mathcal{D}<p_1,\grave{s}_1,h,<p_2,\grave{s}_1,h,\grave{c}_1>>)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) =$ (2× def. $\mathcal{D}_C$)
$N[\![p_1]\!](\mathcal{D}\grave{s}_1)\ h\{N[\![p_2]\!](\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)\}(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) =$ (def. N)
$N[\![p_1 \& p_2]\!](\mathcal{D}\grave{s}_1)h(\mathcal{D}\grave{c}_1)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s}) =$ (def. $\mathcal{D}_C$)
$(\mathcal{D}<p_1 \& p_2,\grave{s}_1,h,\grave{c}_1>)(\mathcal{D}\grave{a})(\mathcal{D}\grave{q})(\mathcal{D}\grave{n})(\mathcal{D}\grave{s})$. □

COROLLARY 5.8. For all p, h and $\grave{s}$, we have

$$O[\![p]\!]\ h\ \grave{s} \neq \perp \ \Rightarrow\ \mathcal{D}(O[\![p]\!]\ h\ \grave{s}) = M[\![p]\!]\ h\ (\mathcal{D}\grave{s}).$$

PROOF. Immediate from Lemma 5.7. □

The other case to be taken care of is the case that $\mathcal{O}[\![p]\!]\ h\ \check{s} = \bot$. We will show in the sequel that this case cannot occur, that is that evaluation of any machine configuration always terminates. It is sufficient to show that there exists a complexity measure $\mathcal{C}$ on machine configurations such that the function *step* decreases this measure for all configurations which are not final.

<u>LEMMA 5.9</u>. If there exists a function $\mathcal{C}$: M $\to$ N such that for all m which are not final we have $\mathcal{C}(m) > \mathcal{C}(step\ m)$; then for all p, h and $\check{s}$ we have that $\mathcal{O}[\![p]\!]\ h\ \check{s} \neq \bot$.

<u>PROOF</u>. We give two proofs depending on which definition of $\mathcal{P}$ is chosen.

1 (The fixed point definition). It is a well known result that $\mathcal{P} = \bigsqcup_i \phi_i$, where $\phi_0 = \lambda m\cdot\bot$ and $\phi_{i+1} = \lambda m\cdot final(m) \to m,\ \phi_i(step\ m)$. The following property will now be proved by induction on i:

$$(m\neq\bot \wedge \phi_i(m) = \bot) \Rightarrow \mathcal{C}[\![m]\!] \geq i. \qquad \ldots (*)$$

<u>Basis</u>. Trivial.

<u>Induction step</u>. Suppose $\phi_{i+1}(m) = \bot$. This implies that m is not final, so we have $\phi_{i+1}(m) = \phi_i(step\ m)$. The induction hypothesis gives that $\mathcal{C}[\![step\ m]\!] \geq i$ and the property of $\mathcal{C}$ yields that $\mathcal{C}[\![m]\!] > \mathcal{C}[\![step\ m]\!] \geq i$ and therefore $\mathcal{C}[\![m]\!] \geq i+1$.

Having proved (*) we now remark that $(\bigsqcup_i \phi_i)(m) = \bot \Rightarrow \forall i\colon \phi_i(m) = \bot \Rightarrow \forall i\colon \mathcal{C}[\![m]\!] \geq i$ which is clearly impossible.

2 (The row definition). If $\mathcal{P}(m) = \bot$ then there exists an infinite row $m = m_1,\ m_2 = step(m_1),\ldots$ with all m_i not final. That is, there exists an infinite row $m_1, m_2, \ldots$ for which $\mathcal{C}[\![m_i]\!] > \mathcal{C}[\![m_{i+1}]\!]$, and this is not possible because for all m_i we have $\mathcal{C}[\![m_i]\!] \geq 0$. □

The rest of this chapter will be devoted to a definition of a function $\mathcal{C}$ with the desired property. We use the following observations.

1. A machine configuration $m = \langle p,s',h,c,a,q,n,s\rangle$ consists in essence of
   a. A row of pattern components, namely p and the components in the list c.
   b. An alternative a, which is in essence a list, the elements of which are again rows of pattern components.

Combining a. and b., we can view a machine configuration as a *list* $<r_1,\dots,r_n>$ of *rows of pattern components* ($r_i = <p_{i_1},\dots,p_{i_k}>$, for some k).

2. Operations, as given by the function *step*, that change the above list are the following:
   a. Operations which change the list into one which is smaller by one element. These are the operations that correspond to a failure in the pattern match. A failure causes backtracking, which amounts to popping a new element from the stack *a*. From this we can conclude that C must be a function that is strictly increasing in the length of the list.
   b. Operations which take one element from the first row of the list. These correspond to cases in which the match succeeds immediately, for instance $\underline{nil}$, $\bar{n}\$\$v$, $\bar{n}..v$, h' (if the match succeeds). The effect is that the first element is taken from the subsequent *c*.
   We conclude that C must be a strictly increasing function of the length of the first row in the list.
   c. Operations, corresponding to patterns $p\$v$, $p.v$ and $p_1\&p_2$, that add an element to the first row of the list. For these operations the following must hold: $C(<<p_1\&p_2,\dots>,\dots>) > C(<<p_1,p_2,\dots>,\dots>)$.
   d. Operations, corresponding to $p_1 \vee p_2$ which enlarge the list by one element. The following must hold: $C(<<p_1 \vee p_2>^{\wedge}rest>,\dots>) >$ $C(<<p_1>^{\wedge}rest,<p_2>^{\wedge}rest,\dots>)$.

If we now define C(list of rows) $= C(<r_1,\dots,r_n>) = C(r_1)+\dots+C(r_n)$ then the property required in a. is satisfied, provided $C(r_i) > 0$. If we take $C(r) = C(<p_1,\dots,p_k>) = C(p_1)\times\dots\times C(p_p)$, then also the property from b. holds, provided $C(p) > 1$. Finally, we can meet the restrictions posed in c. and d. by taking $C(p_1\&p_2) = C(p_1)\times C(p_2) + 1$, and $C(p_1 \vee p_2) = C(p_1)+C(p_2) + 1$, respectively.

The above considerations are formalized in the next definition.

<u>DEFINITION 5.10</u> (C).

1. ($C(p)$). $C(h) = C(\underline{nil}) = C(\underline{abort}) = C(\underline{fail}) = C(n\$\$v) = C(n..v) = 2$
   $C(v) = 3$
   $C(p\$v) = C(p.v) = 3\times C(p)$
   $C(*p) = C(p) + 1$

$C(p_1 \vee p_2) = C(p_1)+C(p_2) + 1$
$C(p_1 \& p_2) = C(p_1)\times C(p_2) + 1.$

2. $(C(c))$. $C(\text{<READY>}) = 2$
$C(<p,s_1,h,c>) = C(p)\times C(c).$

3. $(C(a))$. $C(\text{<FAIL>}) = 1$
$C(c^\wedge <a,q,n>) = C(c)+C(a).$

4. $(C(m))$. $C(m) = 1$ if $final(m)$ holds
$C(c^\wedge <a,q,n,s>) = C(c)+C(a).$

From this definition the following can be established.

LEMMA 5.11.
1. $\forall p$: $C(p) \geq 2$
2. $\forall c$: $C(c) \geq 2$
3. $\forall a$: $C(a) \geq 1$ and $C(a) = 1 \iff a = \text{<FAIL>}$
4. $C(<p,s_1,h,c,a,q,n,s>) = C(p)\times C(c)+C(a)$
5. $\forall m$: $C(m) \geq 1$ and $C(m) = 1 \iff final(m)$.

PROOF. Easy. □

This lemma can be used to prove the result that we were up to:

LEMMA 5.12. $\neg\, final(m) \Rightarrow C(step\ m) < C(m)$.

PROOF. By cases and easy. The proof has been done informally in the remarks preceding Definition 5.10. □

6. ANOTHER OPERATIONAL SEMANTICS

This chapter shows what the consequences can be for the equivalence proof as given in Chapter 5, if another operational semantics is taken. In this chapter we will give an operational semantics in the style of COOK [2], which has also been used in DE BAKKER [1]. We will use definitions from Chapter 3, but occasionally we will feel free to overwrite the definitions from that chapter, for instance the functions O and C will be defined anew here.

In the new approach we chose to separate again the two phases that can be distinguished in the overall matching process, namely the pattern building phase and the matching phase. For this means we first introduce a new syntactic class, the *pattern structures*, which are the results of pattern building, that is which are patterns without free variables.

$o \in \mathit{Patstruct}$, the *pattern structures*.

This class can be defined by $o ::= h \mid \underline{nil} \mid \underline{abort} \mid \underline{fail} \mid o\$v \mid o.v \mid n\$\$v \mid n..v \mid *p \mid o_1 \vee o_2 \mid o_1 \& o_2$.

Notice the clause $*p$ in this definition. Free variables in p will be bound by the $*$-operator.

We have the following lemma on the denotational meaning of pattern structures which should not be a surprise by now:

<u>LEMMA 6.1</u>. $\forall o \in \mathit{Patstruct}: \forall \acute{s}_1, \acute{s}_2: N[\![o]\!]\acute{s}_1 = N[\![o]\!]\acute{s}_2$.

<u>PROOF</u>. Easy, by checking the definition of N. □

This lemma justifies the following definition of the meaning function $L: \mathit{Patstruct} \to \mathit{Str} \to \acute{C} \to \acute{C}$, namely $L[\![o]\!] = N[\![o]\!]\acute{s}$ for some $\acute{s} \in \acute{S}$.

We now introduce the pattern evaluation function E which transforms a pattern expression p, relative to a store $\bar{s}$, into a corresponding pattern structure. This function $E: \mathit{Pat} \to \bar{S} \to \mathit{Patstruct}$ is defined by cases as:

$$
\begin{aligned}
E[\![p]\!]s &= p \quad \text{for } p \equiv h,\ \underline{nil},\ \underline{abort},\ \underline{fail},\ n\$\$v,\ n..v \text{ and } *p' \\
E[\![v]\!]s &= s(v) \\
E[\![p\$v]\!]s &= (E[\![p]\!]s)\$v \\
E[\![p.v]\!]s &= (E[\![p]\!]s).v \\
E[\![p_1\&p_2]\!]s &= (E[\![p_1]\!]s)\ \&\ (E[\![p_2]\!]s) \\
E[\![p_1 \vee p_2]\!]s &= (E[\![p_1]\!]s) \vee (E[\![p_2]\!]s).
\end{aligned}
$$

We have the following lemma, which will be needed in the equivalence proof.

<u>LEMMA 6.2</u>. $\forall p \in \mathit{Pat}\ \forall \bar{s} \in \bar{S}: N[\![p]\!](D\bar{s}) = L[\![E[\![p]\!]\bar{s}]\!]$.

<u>PROOF</u>. By cases (induction on the structure of p). The interesting cases are those where $p \notin \mathit{Patstruct}$. We give two examples: $p \equiv v$, $p \equiv p_1\&p_2$.

1. $N[\![v]\!](\mathcal{D}\tilde{s}) = N[\![(\mathcal{D}\tilde{s})(\!|v|\!)]\!](\mathcal{D}\tilde{s})$ = (by Lemma 5.6) $N[\![\tilde{s}(\!|v|\!)]\!](\mathcal{D}\tilde{s})$ = $L[\![\tilde{s}(\!|v|\!)]\!]$ because $\tilde{s}(\!|v|\!)$ is an element of Str and therefore of $\mathit{Patstruct}$. Now, for the same reason, we have by the definition of E that $E[\![\tilde{s}(\!|v|\!)]\!] = \tilde{s}(\!|v|\!)$, and we are ready.
2. $N[\![p_1\&p_2]\!](\mathcal{D}\tilde{s})\ h\ \tilde{c} = N[\![p_1]\!](\mathcal{D}\tilde{s})\ h\ \{N[\![p_2]\!](\mathcal{D}\tilde{s})\ h\ \tilde{c}\}$ = (ind.)
   $= L[\![E[\![p_1]\!]\tilde{s}]\!]\ h\ \{L[\![E[\![p_2]\!]\tilde{s}]\!]\ h\ \tilde{c}\}$ = (def. L)
   $= N[\![E[\![p_1]\!]\tilde{s}]\!]\ \tilde{s}_1\ h\ \{N[\![E[\![p_2]\!]\tilde{s}]\!]\ \tilde{s}_1\ h\ \tilde{c}\}$ = (def. N)
   $= N[\![(E[\![p_1]\!]\ \tilde{s})\ \&\ (E[\![p_2]\!]\tilde{s})]\!]\ \tilde{s}_1\ h\ \tilde{c}$ = (def. L and E)
   $L[\![E[\![p_1\&p_2]\!]\tilde{s}]\!]\ h\ \tilde{c}$. □

We will next give the new operational semantics. The operational meaning function $\mathcal{O}$ (which has again functionality $\mathcal{O}$: $\mathit{Pat} \to \mathit{Str} \to \tilde{S} \to \tilde{R}$) is defined in terms of an auxiliary function P. The function P takes (among others) a pattern structure and delivers a finite row of intermediate results which are triples. Each triple consists of a cursor position (a numeral), a store and a queue of conditional assignments accumulated. Such a row of intermediate results can be seen as the trace left by the pattern matching process. We thus need the following definition:

$i \in I = (\mathit{Num} \cup \{\text{FAIL},\text{ABORT}\}) \times \tilde{S} \times \tilde{L}$, the class of *intermediate results*.

We furthermore define the tail function κ which takes the last element of a list: $\kappa\langle e_1,\ldots,e_n\rangle = e_n$.

We now define the function P: $\mathit{Patstruct} \to \mathit{Str} \to I \to I^+$ inductively as follows:

1. $P[\![h']\!]\ h\ \langle n,s,q\rangle = \begin{cases}\langle \mathcal{D}^{-1}\ n',s,q\rangle & \text{if } h' \equiv h[\mathcal{D}n{:}n'] \\ \langle \text{FAIL},s,q\rangle & \text{otherwise}\end{cases}$
2. $P[\![\underline{nil}]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle$
3. $P[\![\underline{fail}]\!]\ h\ \langle n,s,q\rangle = \langle \text{FAIL},s,q\rangle$
4. $P[\![abort]\!]\ h\ \langle n,s,q\rangle = \langle \text{ABORT},s,q\rangle$
5. $P[\![o\$v]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle^\wedge P[\![o\&(n\$\$v)]\!]\ h\ \langle n,s,q\rangle$
6. $P[\![o.v]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle^\wedge P[\![o\&(n..v)]\!]\ h\ \langle n,s,q\rangle$
7. $P[\![n'\$\$v]\!]\ h\ \langle n,s,q\rangle = \langle n,s^\wedge\langle v,h[\mathcal{D}n'{:}\mathcal{D}n]\rangle,q\rangle$
8. $P[\![n'..v]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q^\wedge\langle v,h[\mathcal{D}n'{:}\mathcal{D}n]\rangle\rangle$

9. $P[\![o_1 \vee o_2]\!]\ h\ \langle n,s,q\rangle =$
   $n' \neq \text{FAIL} \rightarrow \langle n,s,q\rangle^\wedge P[\![o_1]\!]\ h\ \langle n,s,q\rangle, \langle n,s,q\rangle^\wedge P[\![o_1]\!]\ \text{h}\ \langle \text{n,s,q}\rangle^\wedge P[\![o_2]\!]\ h\ \langle n,s',q\rangle,$
   where $\langle n',s',q\rangle = \kappa(P[\![o_1]\!]\ h\ \langle n,s,q\rangle)$

10. $P[\![*p]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle^\wedge P[\![E[\![p]\!]s]\!]\ h\ \langle n,s,q\rangle$

11. $P[\![\bar{o}\&o_2]\!]\ h\ \langle n,s,q\rangle =$ (where $\bar{o} \equiv h'$, $\underline{nil}$, $\underline{abort}$, $\underline{fail}$, $n''\$\$v$ or $n''..v$)
   $n' = \text{FAIL,ABORT} \rightarrow \langle n,s,q\rangle^\wedge P[\![\bar{o}]\!]\ h\ \langle n,s,q\rangle,$
   $\langle n,s,q\rangle^\wedge P[\![\bar{o}]\!]\ h\ \langle n,s,q\rangle^\wedge P[\![o_2]\!]\ h\ \langle n',s',q'\rangle,$
   where $\langle n',s',q'\rangle = \kappa(P[\![\bar{o}]\!]\ h\ \langle n,s,q\rangle)$

12. $P[\![(o_1\&o_2)\ \&\ o_3]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle^\wedge P[\![o_1\ \&\ (o_2\&o_3))]\!]\ h\ \langle n,s,q\rangle$

13. $P[\![(o_1 \vee o_2)\ \&\ o_3]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle^\wedge P[\![(o_1\&o_3) \vee (o_2\&o_3)]\!]\ h\ \langle n,s,q\rangle$

14. $P[\![(o_1\$v)\ \&\ o_2]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle^\wedge P[\![o_1\ \&\ ((n\$\$v)\ \&\ o_2)]\!]\ h\ \langle n,s,q\rangle$

15. $P[\![(o_1.v)\ \&\ o_2]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle^\wedge P[\![o_1\ \&\ ((n..v)\ \&\ o_2)]\!]\ h\ \langle n,s,q\rangle$

16. $P[\![(*p)\ \&\ o_2]\!]\ h\ \langle n,s,q\rangle = \langle n,s,q\rangle^\wedge P[\![(E[\![p]\!]s)\ \&\ o_2]\!]\ h\ \langle n,s,q\rangle.$

Remarks.

The essential difference with the definition of *step* in Chapter 3 is that here we do not use explicit stacks (c and a). The alternatives remaining are remembered implicitly as can be seen from clause 9: matching against $o_1 \vee o_2$ amounts to matching against o_1 if this match succeeds or is aborted. Otherwise it is the same as matching against o_1 and afterwards against o_2 starting with the correct intermediate result $\langle n,s',q\rangle$.

The subsequents to be applied later are in principle retained in the pattern component itself. Clauses 11 through 16 all deal with patterns of the form $o\ \&\ o'$. Clause 11 ($o \equiv \bar{o}$) gives the case where o does not have implicit alternatives which means that no backtracking to o is possible. In that case matching against o is tried, and we go on if this match succeeds. In all other cases (12 - 16) we have to find out which elementary pattern component has to be matched against first. We solved this by first decomposing the first operand of $o\ \&\ o'$ until an elementary pattern is reached. For instance the pattern structure $(((h_1 \vee h_2)\$v)\ \&\ \underline{fail})\ \&\ o'$ will be rewritten as follows (where we assume that matching starts with cursor position given by the numeral n):

$(((h_1 \vee h_2)\$v)\ \&\ \underline{fail})\ \&\ o' \rightarrow$	(clause 12)
$((h_1 \vee h_2)\$v)\ \&\ (\underline{fail}\ \&\ o') \rightarrow$	(clause 14)
$(h_1 \vee h_2)\ \&\ ((n\$\$v)\ \&\ (\underline{fail}\ \&\ o')) \rightarrow$	(clause 13)

$(h_1\ \&\ ((n\$\$v)\ \&\ (\underline{fail}\ \&\ o'))) \vee (h_2\ \&\ ((n\$\$v)\ \&\ (\underline{fail}\ \&\ o')).$

Now we use clause 9, and first investigate the first operand of this disjunction which is $(h_1 \,\&\, ((n\$\$v) \,\&\, (\underline{fail} \,\&\, o')))$. On this pattern structure we then apply clause 11, clause 1, etc.

Notice also clause 10 and 16 of the definition. If, while matching, the scanner encounters an $*$-operator, first the corresponding pattern component is evaluated using $\mathcal{E}$, before proceeding.

The claim on the functionality of $\mathcal{P}$, in particular that $\mathcal{P}$ yields values in I^+, and also the statement that the above definition is an inductive one, has to be justified. We do this by presenting a complexity measure $\mathcal{C}$ on pattern structures such that all structures occurring in the right-hand sides of the clauses of the definition of $\mathcal{P}$ have smaller $\mathcal{C}$-values than the o's in the corresponding left-hand sides. The following function $\mathcal{C}$, defined on Pat by structural induction, does the job:

$$\begin{aligned}
&\mathcal{C}[\![h]\!] = \mathcal{C}[\![v]\!] = \mathcal{C}[\![\underline{nil}]\!] = \mathcal{C}[\![\underline{abort}]\!] = \mathcal{C}[\![\underline{fail}]\!] = \mathcal{C}[\![n\$\$v]\!] = \mathcal{C}[\![n..v]\!] = 1\\
&\mathcal{C}[\![p\$v]\!] = \mathcal{C}[\![p.v]\!] = 2\mathcal{C}[\![p]\!] + 2\\
&\mathcal{C}[\![*p]\!] = \mathcal{C}[\![p]\!] + 1\\
&\mathcal{C}[\![p_1\&p_2]\!] = 2\mathcal{C}[\![p_1]\!] + \mathcal{C}[\![p_2]\!]\\
&\mathcal{C}[\![p_1 \vee p_2]\!] = \max\{\mathcal{C}[\![p_1]\!], \mathcal{C}[\![p_2]\!]\} + 1.
\end{aligned}$$

We are now ready to define the function $\mathcal{O}$ with functionality $\mathcal{O}\colon \mathit{Pat} \to \mathit{Str} \to S \to R$ as follows:

$$\begin{aligned}
&\mathcal{O}[\![p]\!]\; h\; s = (n' = \text{ABORT},\text{FAIL}) \to \langle n',s'\rangle, \langle n', s'{}^\wedge q'\rangle,\\
&\qquad \text{where } \langle n',s',q'\rangle = \kappa(\mathcal{P}[\![\mathcal{E}[\![p]\!]s]\!]\; h\; \langle 0,s,\langle\rangle\rangle\rangle.
\end{aligned}$$

In order to be able to prove an equivalence result similar to the one in Chapter 5, we need some auxiliary facts:

<u>LEMMA 6.3.</u>

1. $\mathcal{N}[\![(p_1\&p_2)\;\&\;p_3]\!] = \mathcal{N}[\![p_1\;\&\;(p_2\&p_3)]\!]$
2. $\mathcal{N}[\![(p_1 \vee p_2)\;\&\;p_3]\!] = \mathcal{N}[\![(p_1\&p_3) \vee (p_2\&p_3)]\!]$
3. $\mathcal{N}[\![(p_1\$v)\;\&\;p_2]\!]\; s_1\; h\; c\; a\; q\; n\; s = \mathcal{N}[\![p_1\;\&\;((\mathcal{D}^{-1}n\$\$v)\;\&\;p_2)]\!]\; s_1\; h\; c\; a\; q\; n\; s$
4. $\mathcal{N}[\![(p_1.v)\;\&\;p_2]\!]\; s_1\; h\; c\; a\; q\; n\; s = \mathcal{N}[\![p_1\;\&\;((\mathcal{D}^{-1}n..v)\;\&\;p_2)]\!]\; s_1\; h\; c\; a\; q\; n\; s$
5. $\mathcal{N}[\![(*p_1)\;\&\;p_2]\!]\; s_1\; h\; c\; a\; q\; n(\mathcal{D}\bar{s}) = \mathcal{N}[\![(\mathcal{E}[\![p_1]\!]\bar{s})\;\&\;p_2]\!]\; s_1\; h\; c\; a\; q\; n\; (\mathcal{D}\bar{s})$.

PROOF.

1 and 2. By writing out the respective clauses in the definition of N.

3 and 4. By Lemma 4.1, by result 1 of this lemma, and by the fact that

$N[\![p_1]\!] = N[\![p_2]\!]$ implies $N[\![p_1 \& p_3]\!] = N[\![p_2 \& p_3]\!]$.

5. $N[\![(*p_1) \& p_2]\!]\ s_1\ h\ c\ a\ q\ n\ (\mathcal{D}\tilde{s}) =$ (def. N)

$N[\![*p_1]\!]\ s_1\ h\ \{N[\![p_2]\!]\ s_1\ h\ c\}\ a\ q\ n\ (\mathcal{D}\tilde{s}) =$ (def. N)

$N[\![p_1]\!](\mathcal{D}\tilde{s})\ h\ \{N[\![p_2]\!]\ s_1\ h\ c\}\ a\ q\ n\ (\mathcal{D}\tilde{s}) =$ (Lemma 6.2, 6.1)

$N[\![E[\![p_1]\!]\tilde{s}]\!]\ s_1\ h\ \{N[\![p_2]\!]\ s_1\ h\ c\}\ a\ q\ n\ (\mathcal{D}\tilde{s}) =$ (def. N)

$N[\![(E[\![p_1]\!]\tilde{s}) \& p_2]\!]\ s_1\ h\ c\ a\ q\ n\ (\mathcal{D}\tilde{s})$. □

Now if we want to prove $\mathcal{O}$ and M equivalent it appears that we have to formulate a rather complicated induction hypothesis relating N and P. This is due to the fact that in the operational definition of this chapter no counterparts of the entities c and a from the denotational definition exist. We have to capture the effects that subsequents and alternatives may have by formulating the following induction hypothesis. The main trick is that we capture the effect of the alternative a in the denotational definition by quantifying over all alternatives.

LEMMA 6.4. For all o, h, $\tilde{n}$, $\tilde{s}$ and $\tilde{q}$ we have

$$P[\![o]\!]\ h\ \langle\tilde{n},\tilde{s},\tilde{q}\rangle \cong \lambda a\cdot L[\![o]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$$

where $list \cong \phi$ iff

$$\left[\begin{array}{lll} \underline{\text{either}} & \kappa(list) = \langle\text{FAIL},\tilde{s}_1,\tilde{q}_1\rangle & \text{and } \phi = \lambda a\cdot a(\mathcal{D}\tilde{s}_1) \\ \underline{\text{or}} & \kappa(list) = \langle\text{ABORT},\tilde{s}_1,\tilde{q}_1\rangle & \text{and } \phi = \lambda a\cdot\langle\text{ABORT},(\mathcal{D}\tilde{s}_1)\rangle \\ \text{or} & \kappa(list) = \langle\tilde{n}_1,\tilde{s}_1,\tilde{q}_1\rangle & \text{and } \phi = \lambda a\cdot\langle\mathcal{D}\tilde{n}_1,(\mathcal{D}\tilde{q}_1)(\mathcal{D}\tilde{s}_1)\rangle \end{array}\right].$$

PROOF. By induction on the C-complexity of o. We have to distinguish all cases as occurring in the definition of P which is tedious. So we give a few typical examples. We define $lhs = \kappa(P[\![o]\!]\ h\ \langle\tilde{n},\tilde{s},\tilde{q}\rangle)$ and $rhs = \lambda a\cdot L[\![o]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s})$.

2. (nil) $lhs = \langle\tilde{n},\tilde{s},\tilde{q}\rangle$ and $rhs = \lambda a\cdot\langle\mathcal{D}\tilde{n},(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{s})\rangle$

3. (fail) $lhs = \langle\text{FAIL},\tilde{s},\tilde{q}\rangle$ and $rhs = \lambda a\cdot a(\mathcal{D}\tilde{s})$

8. $(n'..v)$ We have $lhs = <\tilde{n},\tilde{s},\tilde{q}^{\wedge}<v,h[\mathcal{D}n':\mathcal{D}\tilde{n}]>>$ and

$rhs = \lambda a\cdot L[\![n'..v]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$

$\lambda a\cdot ready\ a\ (\lambda\tilde{s}\cdot((\mathcal{D}\tilde{q})\tilde{s})\{h[\mathcal{D}n':\mathcal{D}\tilde{n}]/v\})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$

$<\mathcal{D}\tilde{n},((\mathcal{D}\tilde{q})(\mathcal{D}\tilde{s}))\{h[\mathcal{D}n':\mathcal{D}\tilde{n}]/v\}>$.

So we have to prove that this is equal to $(\mathcal{D}(\tilde{q}^{\wedge}<v,h[\mathcal{D}n':\mathcal{D}\tilde{n}]>))(\mathcal{D}\tilde{s})$ and this is true by the definition of $\mathcal{D}_Q$.

9. $(o_1 \vee o_2)$. Let $<\tilde{n}_2,\tilde{s}_2,\tilde{q}_2> = \kappa(P[\![o_1]\!]\ h\ <\tilde{n},\tilde{s},\tilde{q}>)$.

A. n_2 = ABORT.

Then $lhs = <\text{ABORT},\tilde{s}_2,\tilde{q}_2>$ and by induction we have for all a':

$L[\![o_1]\!]\ h\ ready\ a'\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) = <\text{ABORT},\mathcal{D}\tilde{s}_2>$.

This holds in particular for $a' = L[\![o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})$ and we thus get for all a: $L[\![o_1 \vee o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) = <\text{ABORT},\mathcal{D}\tilde{s}_2>$.

B. $n_2 \in Num$.

The argument is similar to that in case A.

C. n_2 = FAIL.

We have $lhs = \kappa(P[\![o_2]\!]\ h\ <\tilde{n},\tilde{s}_2,q>)$.

By induction we have for all a' that

$L[\![o_1]\!]\ h\ ready\ a'(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) = a'(\mathcal{D}\tilde{s}_2)$. This holds in particular for $a' = L[\![o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})$ and we get

$rhs = \lambda a\cdot L[\![o_1 \vee o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$

$\lambda a\cdot L[\![o_1]\!]\ h\ ready\ \{L[\![o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})\}(\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}) =$

$\lambda a\cdot L[\![o_2]\!]\ h\ ready\ a\ (\mathcal{D}\tilde{q})(\mathcal{D}\tilde{n})(\mathcal{D}\tilde{s}_2)$.

Now we can apply the induction hypothesis, for $C[\![o_2]\!] < C[\![o_1 \vee o_2]\!]$.

12. $((o_1 \& o_2) \& o_3)$. Use Lemma 6.3.1, and the induction hypothesis (notice that $C[\![o_1 \& (o_2 \& o_3)]\!] < C[\![(o_1 \& o_2) \& o_3]\!]$.

16. $((*p) \& o_2)$. Use Lemma 6.3.5 and the induction hypothesis. □

THEOREM 6.5. For all p, h and $\tilde{s}$ we have $\mathcal{D}(O[\![p]\!]\ h\ \tilde{s}) = M[\![p]\!]\ h\ (\mathcal{D}\tilde{s})$.

PROOF. We have $M[\![p]\!]\ h\ (\mathcal{D}\tilde{s}) = N[\![p]\!](\mathcal{D}\tilde{s})\ h\ ready\ fail\ (\lambda s\cdot s)\ 0\ (\mathcal{D}\tilde{s}) =$
$= L[\![E[\![p]\!]\tilde{s}]\!]\ h\ ready\ fail\ (\lambda s\cdot s)\ 0\ (\mathcal{D}\tilde{s})$ by Lemma 6.2.

Let $\kappa(P[\![E[\![p]\!]\tilde{s}]\!]\ h\ <0,\tilde{s},<>> = <\tilde{n}_1,\tilde{s}_1,\tilde{q}_1>$. There are three cases.

1. n_1 = FAIL. By Lemma 6.4.: $M[\![p]\!]\ h\ (\mathcal{D}\tilde{s}) = fail\ (\mathcal{D}\tilde{s}_1) = <\text{FAIL},\mathcal{D}\tilde{s}_1>$.
By the definition of O we have $O[\![p]\!]\ h\ \tilde{s} = <\text{FAIL},\tilde{s}_1>$.

2. If n_1 = ABORT, then by Lemma 6.4 we have that $M[\![p]\!]\ h\ (\mathcal{D}\tilde{s}) = <\text{ABORT},\mathcal{D}\tilde{s}_1>$ and by the definition of O we have $O[\![p]\!]\ h\ \tilde{s} = <\text{ABORT},\tilde{s}_1>$.

3. If $n_1 \in \mathit{Num}$ then Lemma 6.4 gives $\mathcal{M}[\![p]\!]\ h\ (\mathcal{D}\tilde{s}) = <\mathcal{D}\tilde{n}_1,(\mathcal{D}\tilde{q}_1)(\mathcal{D}\tilde{s}_1)>$, while the definition of $\mathcal{O}$ yields $\mathcal{O}[\![p]\!]\ h\ \tilde{s} = <\tilde{n}_1,\tilde{s}_1\hat{\ }\tilde{q}_1>$. Now by Lemma 5.2: $\mathcal{D}<\tilde{n}_1,\tilde{s}_1\hat{\ }\tilde{q}_1> = <\mathcal{D}\tilde{n}_1,(\mathcal{D}\tilde{q}_1)(\mathcal{D}\tilde{s}_1)>$ and we are ready. □

7. CONCLUDING REMARKS AND ACKNOWLEDGEMENTS

This report presents the first results of a project in which we aim to study various semantical aspects of the matching process in SNOBOL4. The next step to be taken is to allow patterns as values of variables, instead of strings as was the case here. This will lead to a (denotational) store S which will be a function from variables to patterns, where patterns are modelled by functions which describe (amongst others) store transformations. This suggests a reflexive (circular) definition of the domain of stores, and an equivalence proof like the one given here will be much harder to construct.

This is why we chose to do some "ground work" first, and this paper presents the results of it. We chose the SNOBOL subset such that all essential aspects of pattern matching are reflected in it, apart from the idea that patterns can be values of variables.

The denotational semantics given here should be compared with the one given by TENNENT [9] which is far more complicated due to the fact that a much larger subset of SNOBOL4 is involved here. Our semantics can be viewed as a simplification of Tennent's, resulting in a semantics that describe the matching process clearly with no more tools and complications than needed.

In Chapter 6 we showed that one has to be careful in designing an operational semantics, if one wishes to prove an equivalence result. The semantics of Chapter 3 is inspired by the operational semantics in STOY [8]. We borrowed his idea to carefully provide for each denotational notion a corresponding operational notion. For instance our operational semantics uses a class of subsequents and a class of alternatives which correspond to the denotational domains C and A. This made the equivalence proof manageable, as can be seen when one compares the proof in Chapter 5 with the one in Chapter 6 where an operational semantics was used which was less carefully designed.

Apart from Stoy's, the papers by GIMPEL [3] and PAGAN [6] should be mentioned. They provided many useful details about the peculiarities of the SNOBOL language.

Finally, I like to mention Jaco de Bakker, who has read an earlier version of this paper and who came up with useful comments, and also Ruurd Kuiper with whom I had fruitful discussions on the topics treated here.

REFERENCES

[1] BAKKER, J.W. DE, *Mathematical theory of program correctness,* Prentice Hall Int. (1980).

[2] COOK, S.A., *Soundness and completeness of an axiom system for program verification,* SIAM J. on Computing, Vol. 7, nr. 1, pp. 70-90 (1978).

[3] GIMPEL, J.F., *A theory of discrete patterns and their implementation in SNOBOL4,* Comm. of the ACM, Vol. 16, nr. 2, pp. 91-100 (1973).

[4] LANDIN, P.J., *The mechanical evaluation of expressions,* Computer J., Vol. 6, nr. 4, pp. 308-320 (1964).

[5] MILNE, R. & C. STRACHEY, *A theory of programming language semantics,* Chapman and Hall, London and Wiley, New York, 2 vols. (1976).

[6] PAGAN, F.G., *Formal semantics of a SNOBOL4 subset,* Computer Languages, Vol. 3, pp. 13-30 (1978).

[7] STOY, J.E., *Denotational semantics - the Scott-Strachey approach to programming language theory,* M.I.T. Press, Cambridge, Mass. (1977).

[8] STOY, J.E., *The congruence of two programming language definitions,* to appear.

[9] TENNENT, R.D., *Mathematical semantics and design of programming languages,* University of Toronto, Technical Report nr. 59 (1973).

chapter 4

GOTO STATEMENTS: SEMANTICS AND DEDUCTION SYSTEMS

Acta Informatica 15, 385–424 (1981)

Acta Informatica
© Springer-Verlag 1981

Goto Statements: Semantics and Deduction Systems

Arie de Bruin

Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Summary. A simple language containing **goto** statements is presented, together with a denotational and operational semantic for it. Equivalence of these semantical descriptions is proven.

Furthermore, soundness and completeness of a Hoare-like proof system for the language is shown. This is done in two steps. Firstly, a proof system is given and validity is defined using (a variant of) direct semantics. In this case soundness and completeness proofs are relatively easy. After that, a proof system is given which is more in the style of the one by Clint and Hoare [8], and validity in this system is defined using continuation semantics. This validity definition is then related to validity in the first system and, using this correspondence, soundness and completeness for the second system is proven.

1. Introduction

In this report we present several ways of looking at the meaning of **goto** statements. We define a simple language containing **goto** statements, and present an operational definition of its semantics in the sense of [9]. We also give a denotational semantics, using the concept of continuations [17]. Furthermore we prove that these definitions are equivalent.

After that we turn our attention towards a Hoare-like deduction system, as proposed in [8], for proving partial correctness of programs of our language. It appears to be surprisingly complicated to justify this system. The essential rule in the deduction system is (for programs with one label only)

$$\frac{\{p\}\ \mathbf{goto}\ L\{\mathbf{false}\} \vdash \{p_1\}\ A_1\{p\} \qquad \{p\}\ \mathbf{goto}\ L\{\mathbf{false}\} \vdash \{p\}\ A_2\{p_2\}}{\vdash \{p_1\}\ A_1;\ L\colon A_2\{p_2\}}$$

and the unusual assumption $\{p\}$ **goto** $L\{$**false**$\}$ already gives an indication of possible complications. The main problem is how validity of the construct $\{p\}\,A\,\{q\}$ has to be defined.

If we investigate how the inference rule given above will be used in correctness proofs, we observe that the assumption $\{p\}$ **goto** $L\{$**false**$\}$ is used as a trick to indicate that p always holds before execution of **goto** L. Or, stated another way, the assertion p in the assumption serves as a so called *label invariant:* if we want to prove partial correctness of a program S which contains a label L, then we can use the assumption $\{p\}$ **goto** $L\{$**false**$\}$ in the proof to describe that p holds whenever label L is encountered during evaluation of S. Thus the introduction of an assumption like $\{p\}$ **goto** $L\{$**false**$\}$ in a proof only serves the purpose to indicate what the label invariant at L will be.

This report gives two variants of the deduction system and the above observations are used in the first one. Here there are no assumptions, the label invariants needed are stated explicitly within the formulae of the system. These formulae will have the form

$$\langle L_1 : p_1, \ldots, L_n : p_n \mid \{p\}\,A\,\{q\}\rangle,$$

where p_i is the invariant corresponding to label L_i. Validity of a formula like this one has to be defined in terms of the meaning of statement A occurring in it. Things become too complicated if we use the customary denotational definition with continuations and environments. Techniques in the spirit of "continuation removal" [13] are used to define the meaning of statements such that a definition of validity is possible which is both perspicuous and useful. After that, soundness and completeness of the deduction system will be proved.

Once this result has been established we investigate a deduction system like the one given by Clint and Hoare. We give a definition of validity of formulae like $\{p\}\,A\,\{q\}$ using the ordinary continuation semantics. This definition resembles closely the one given in [13]. Furthermore this definition of validity is such that

$$\{p_1\}\ \mathbf{goto}\ L_1\ \{\mathbf{false}\}, \ldots, \{p_n\}\ \mathbf{goto}\ L_n\{\mathbf{false}\} \models \{p\}\,A\,\{q\}$$

holds, if and only if in the other system the formula

$$\langle L_1 : p_1, \ldots, L_n : p_n \mid \{p\}\,A\,\{q\}\rangle$$

is valid. This result will then be used to prove soundness and completeness for the second deduction system.

This two level approach has the following advantages. In the first variant of the system we take only those elements of the Clint-Hoare system into account that are really necessary. This has as a consequence that the definition of validity and the arguments in the soundness and completeness proofs are as perspicuous as possible. Though straightforward proofs of these properties for the second system must essentially be the same as the ones for the first variant, they are bound to be obscured through all additional details which we have to deal with. The way we handle this problem is to separate the "essential proof" from the "additional details".

The rules and axioms in the second system are just like the ones in other Hoare-like system, and we can combine these into one system quite easily. Using the validity definition for the second variant of the system, as given in this report, it must possible to combine the results stated here with analogous results concerning other language constructs (such as **while** statements, recursive procedures and the like; cf. de Bakker's monograph [5], see [1, 2], and [6]).

2. Syntax

We use the following classes of symbols:

$\mathscr{V}ar$, the (infinite) class of *variables* with typical elements x, y, z. We assume this class to be ordered

$\mathscr{L}var$, the class of *label variables* with typical element L

$\mathscr{F}sym=\{fu_1, \ldots, fu_m\}$, the class of *function symbols*. We denote the arity of fu_i by arf_i

$\mathscr{R}sym=\{re_1, \ldots, re_n\}$, the class of *relation symbols*. The arity of re_i is denoted by arr_i.

Next, using a self-explanatory variant of BNF, we define the classes $\mathscr{B}exp$ (*boolean expressions*) with typical element b, $\mathscr{E}xp$ (*expressions*) with typical elements s, t, $\mathscr{S}tat$ (*statements*) with typical element A, and $\mathscr{P}rog$ (*programs*) with typical element S:

$\mathscr{B}exp$ $\quad b ::= \textbf{true} \mid b_1 \vee b_2 \mid \neg b \mid re_1(s_1, \ldots, s_{arr_1}) \mid \ldots \mid re_n(s_1, \ldots, s_{arr_n})$

$\mathscr{E}xp$ $\quad s ::= x \mid fu_1(s_1, \ldots, s_{arf_1}) \mid \ldots \mid fu_m(s_1, \ldots, s_{arf_m})$

$\mathscr{S}tat$ $\quad A ::= x := s \mid (A_1; A_2) \mid \textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi} \mid \textbf{goto } L$

$\mathscr{P}rog$ $\quad S ::= L\colon A \mid L\colon A; S$

with the additional requirement: if $L_1 : A_1; \ldots; L_n : A_n$ is a program, then all labels L_i are different.

The symbols fu_i and re_i are the primitive function and relation symbols. We did not specify them further, because we do not wish to go into details concerning the basic calculations our programs S can perform. Rather do we want to describe the way programs specify more complex calculations using these primitives as building blocks.

The clause $(A_1; A_2)$ in the definition of $\mathscr{S}tat$ deserves some comment. It is usual to omit parentheses in cases like this one, thus admitting the grammar to be ambiguous. In general there is no problem there, because the meaning of, say $(A_1; A_2); A_3$ and $A_1; (A_2; A_3)$ will be the same. Of course this holds in our case too. However, complications show up in our definition of the operational semantics. For instance, for the auxiliary semantic function $\mathscr{C}omp$ the equality

$$\mathscr{C}omp((A_1; A_2); A_3) = \mathscr{C}omp(A_1; (A_2; A_3))$$

does not hold.

Putting parentheses all over the place is tedious though. We therefore use the convention that the operator ";" associates to the right, which means that $A_1; A_2; \ldots; A_n$ should be read as $(A_1; (A_2; (\ldots; A_n)..))$.

Anticipating the deduction systems of Sects. 6 and 8 we give the definition of the syntactical class $\mathscr{A}ssn$ (the *assertions*) with typical elements p, q.

$$\mathscr{A}ssn\ p ::= \textbf{true}\ |p_1 \vee p_2| \neg p | re_1(s_1, \ldots, s_{arr_1})| \ldots |re_n(s_1, \ldots, s_{arr_n})|\ \exists x[p]$$

It turns out that $\mathscr{A}ssn$ is just a language $\mathscr{L}$ for the first order predicate calculus, based on the classes $\mathscr{F}sym$ and $\mathscr{R}sym$. Furthermore we see that $\mathscr{E}xp$ is exactly the class of the terms of $\mathscr{L}$, and that $\mathscr{B}exp$ is the set of all quantifier free formulae of $\mathscr{L}$.

The rest of this paragraph gives some notational conventions and useful definitions. We use the symbol $\equiv$ to refer to *syntactical identity*, i.e., $B \equiv C$ means that B and C are the same sequence of symbols. The following abbreviations will be used:

$$b_1 \wedge b_2 \equiv \neg(\neg b_1 \vee \neg b_2)$$
$$b_1 \supset b_2 \equiv \neg b_1 \vee b_2$$
$$\textbf{if } b_1 \textbf{ then } b_2 \textbf{ else } b_3 \textbf{ fi} \equiv (b_1 \wedge b_2) \vee (\neg b_1 \wedge b_3)$$
$$\textbf{false} \equiv \neg \textbf{true}$$
$$[L_i : A_i]_{i=1}^n \equiv L_1 : A_1; \ldots; L_n : A_n.$$

We define the property that *a label L occurs in A* inductively by

a) no label occurs in a statement of the form $x := s$

b) L occurs in $(A_1; A_2)$ and in **if** b **then** A_1 **else** A_2 **fi**, if either L occurs in A_1 or L occurs in A_2

c) the only label occurring in **goto** L is L.

Let $S \equiv [L_i : A_i]_{i=1}^n$. We say that

+) *L is declared in S* if $L \equiv L_i$ for some i $(1 \leqq i \leqq n)$

+) *L occurs in S* if either L is declared in S or L occurs in some A_i $(1 \leqq i \leqq n)$

+) *S is normal* if all labels occurring in S are declared in S.

3. Operational Semantics

In our semantics functions will be used abundantly. Often these functions will be of higher order, which means that they have functions as arguments and/or values. In order to keep our notation as clear as possible we first state some conventions on this point.

a) The class of all functions with domain A and range B will be denoted by $(A \rightarrow B)$

b) The class of all partial functions with domain A and range B will be denoted by $(A \xrightarrow[p]{} B)$

c) The convention will be used that "$\rightarrow$" associates to the right. For example, $A \rightarrow B \rightarrow C$ must be read as $A \rightarrow (B \rightarrow C)$

d) We will in general omit parentheses around arguments, using the convention that function application associates to the left. Thus, assuming

$f \in (A \to B \to C \to D)$, $a \in A$, $b \in B$, $c \in C$, for some A, B, C and D, the entity $((f(a))(b))(c)$ can be written as $fabc$

e) The above convention has the following exception: every syntactic entity used as an argument will be enclosed in $[\![\cdot]\!]$-type brackets. This is done to provide a clear distinction between the object language of Sect. 2 and the language used to denote the semantic objects.

As a starting point of our semantical considerations we first discuss the meaning given to the symbols in $\mathcal{F}sym$ and $\mathcal{R}sym$. An *interpretation* $\mathcal{I}$ *of the primitive symbols* is an $(m+n+1)$-tuple $\langle D, \underline{fu}_1, \ldots, \underline{fu}_m, \underline{re}_1, \ldots, \underline{re}_n \rangle$, where

D is a non-empty domain,
$\underline{fu}_i$ is a function in $(D^{arf_i} \to D)$, for $i = 1, \ldots, m$, and
$\underline{re}_i$ is a relation $\subset D^{arr_i}$ $(i = 1, \ldots, n)$.

All semantic functions to be defined will depend on an underlying interpretation of the primitive symbols, though the notation we use won't show this dependence. For instance, the function giving the meaning of the expressions will be denoted by $\mathcal{V}$, instead of $\mathcal{V}_{\mathcal{I}}$ or something like that.

We now choose an arbitrary interpretation $\mathcal{I}$ and assume this interpretation to remain fixed for the rest of this paper (unless we explicitly state otherwise).

A *state* is a valuation of the variables from $\mathcal{V}ar$ in our domain of interpretation D. The set of all states is denoted by Σ, with typical element σ. In principle the meaning of a statement will be a partial function from states to states. The function is partial, due to the possibility of nonterminating computations. We consider it useful not to allow partial functions and therefore include the undefined state $\perp$ in Σ. This leads to the following definition:

$$\Sigma = (\mathcal{V}ar \to D) \cup \{\perp\}.$$

We denote the set of all defined states by Σ_0, i.e. $\Sigma_0 = (\mathcal{V}ar \to D)$.

Let $d \in D$. A *variant* $\sigma\{d/x\}$ *of a state* σ is a state σ' differing from σ only in the variable x, or explicitly

$$\sigma\{d/x\} = \begin{cases} \perp, \text{ if } \sigma = \perp \text{ and otherwise} \\ \sigma' \in \Sigma_0 \text{ such that } \sigma'[\![y]\!] = \begin{cases} \sigma[\![y]\!] & \text{if } x \not\equiv y \\ d & \text{if } x \equiv y. \end{cases} \end{cases}$$

The next syntactic classes to be handled are $\mathcal{B}exp$ and $\mathcal{E}xp$. We will define inductively the semantic functions

$$\mathcal{V}: \mathcal{E}xp \to \Sigma_0 \to D$$
$$\mathcal{W}: \mathcal{B}exp \to \Sigma_0 \to \{ff, tt\}.$$

Note that $\mathcal{V}[\![s]\!]\,\sigma$, and $\mathcal{W}[\![b]\!]\,\sigma$ are not defined for $\sigma = \perp$.

Definition of $\mathcal{V}$.

$\mathcal{V}[\![x]\!]\,\sigma = \sigma[\![x]\!]$

$\mathcal{V}[\![fu_i(s_1, \ldots, s_{arf_i})]\!]\,\sigma = \underline{fu}_i(\mathcal{V}[\![s_1]\!]\,\sigma, \ldots, \mathcal{V}[\![s_{arf_i}]\!]\,\sigma).$

Definition of $\mathscr{W}$.

$\mathscr{W}[\![\textbf{true}]\!]\,\sigma = tt$

$\mathscr{W}[\![b_1 \vee b_2]\!]\,\sigma = tt$, if $\mathscr{W}[\![b_1]\!]\,\sigma = tt$ or $\mathscr{W}[\![b_2]\!]\,\sigma = tt$, and ff otherwise

$\mathscr{W}[\![\neg b]\!]\,\sigma = tt$, if $\mathscr{W}[\![b]\!]\,\sigma = ff$, and ff otherwise

$$\mathscr{W}[\![re_i(s_1, \ldots, s_{arr_i})]\!]\,\sigma = \begin{cases} tt, & \text{if } \langle \mathscr{V}[\![s_1]\!]\,\sigma, \ldots, \mathscr{V}[\![s_{arr_i}]\!]\,\sigma \rangle \in \underline{re}_i \\ ff, & \text{otherwise.} \end{cases}$$

The semantic definitions given above are basic in the sense that they will be the same for the denotational semantics. We now turn to the operational semantics proper.

We want to define the meaning of a statement A as a function that, given an initial state σ as an argument, yields a so-called *computation sequence* τ. Such a computation sequence is a possibly infinite row of states from Σ, the elements of which can be viewed as the successive intermediate states produced by evaluation of the statement A starting in initial state σ. The semantic function that maps statements on their meaning in the above sense will be called $\mathscr{C}omp$.

In order to be able to handle these computation sequences, we present the following definitions:

a) *(Computation Sequences)*

Σ^+ is the class of all non-empty finite sequences $\langle \sigma_0, \ldots, \sigma_n \rangle$ for some $n \geqq 0$, such that $\sigma_i \in \Sigma$ for $i = 0, 1, \ldots, n$

Σ^ω is the class of all infinite sequences $\langle \sigma_0, \sigma_1, \ldots \rangle$, such that $\sigma_i \in \Sigma$ for all $i \in \mathbb{N}$

Σ^∞, with typical element τ, is defined through $\Sigma^\infty = \Sigma^+ \cup \Sigma^\omega$.

b) *(Concatenation)*

Let $\tau_1, \tau_2 \in \Sigma^\infty$. The concatenation of τ_1 and τ_2, notation $\tau_1{}^\frown\tau_2$, is defined by

1) if $\tau_1 = \langle \sigma_0, \ldots, \sigma_n \rangle \in \Sigma^+$ and $\tau_2 = \langle \sigma'_0, \ldots, \sigma'_m \rangle \in \Sigma^+$ then
$\tau_1{}^\frown\tau_2 = \langle \sigma_0, \ldots, \sigma_n, \sigma'_0, \ldots, \sigma'_m \rangle \in \Sigma^+$

2) if $\tau_1 = \langle \sigma_0, \ldots, \sigma_n \rangle \in \Sigma^+$ and $\tau_2 = \langle \sigma'_0, \sigma'_1, \ldots, \rangle \in \Sigma^\omega$ then
$\tau_1{}^\frown\tau_2 = \langle \sigma_0, \sigma_1, \ldots, \sigma_n, \sigma'_0, \sigma'_1, \ldots \rangle \in \Sigma^\omega$

3) if $\tau_1 \in \Sigma^\omega$ then $\tau_1{}^\frown\tau_2 = \tau_1$.

c) *(κ-Function)*

The function $\kappa \in (\Sigma^\infty \to \Sigma)$ is defined by

$$\kappa(\tau) = \begin{cases} \bot, & \text{if } \tau \in \Sigma^\omega \\ \sigma_n, & \text{if } \tau = \langle \sigma_0, \ldots, \sigma_n \rangle \in \Sigma^+. \end{cases}$$

There is a last remark to be made before we give an exact definition of $\mathscr{C}omp$. We must be aware of the fact that A can contain substatements of the form **goto** L, and we should have a way to get to know how evaluation of A proceeds once such a substatement is reached. We therefore supply the function $\mathscr{Comp}$ with an extra argument, namely an element of $\mathscr{Prog}$, meant to provide the "declaration" of the labels occurring in A. $\mathscr{Comp}$ will then have the following functionality:

$$\mathscr{Comp}: \mathscr{Prog} \times \mathscr{Stat} \to \Sigma \xrightarrow[p]{} \Sigma^{\infty}$$

and the computation sequence $\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma$ is meant to be the row of intermediate states appearing during evaluation of A starting in state σ, where the labels are defined by the program S.

Definition ($\mathscr{Comp}$). A. $\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma = \langle \bot \rangle$ if $\sigma = \bot$.

B. $\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma$ for $\sigma \in \Sigma_0$ is defined recursively by

1. $\mathscr{Comp}[\![\langle S, x := s\rangle]\!]\,\sigma = \langle \sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\}\rangle$
2. $\mathscr{Comp}[\![\langle S,$ **if** b **then** A_1 **else** A_2 **fi**$\rangle]\!]\,\sigma$
$$= \begin{cases} \langle\sigma\rangle^\frown \mathscr{Comp}[\![\langle S, A_1\rangle]\!]\,\sigma, & \text{if } \mathscr{W}[\![b]\!]\,\sigma = tt \\ \langle\sigma\rangle^\frown \mathscr{Comp}[\![\langle S, A_2\rangle]\!]\,\sigma, & \text{if } \mathscr{W}[\![b]\!]\,\sigma = ff \end{cases}$$
3. $\mathscr{Comp}[\![\langle S,$ **goto** $L\rangle]\!]\,\sigma$
$$= \begin{cases} \langle\sigma\rangle^\frown \mathscr{Comp}[\![\langle S, A_i; A_{i+1}, \ldots A_n\rangle]\!]\,\sigma, & \text{if } S \equiv [L_1 : A_i]_{i-1}^{n}, \\ \quad \text{and } L \equiv L_i \text{ for some } i,\ 1 \leqq i \leqq n \\ \text{undefined, otherwise} \end{cases}$$
4. $\mathscr{Comp}[\![\langle S, (x := s; A')\rangle]\!]\,\sigma = \langle \sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\}\rangle^\frown \mathscr{Comp}[\![\langle S, A'\rangle]\!]\,(\sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\})$
5. $\mathscr{Comp}[\![\langle S, ((A''; A'''); A')\rangle]\!]\,\sigma = \langle\sigma\rangle^\frown \mathscr{Comp}[\![\langle S, (A''; (A'''; A'))\rangle]\!]\,p$
6. $\mathscr{Comp}[\![\langle S,($**if** b **then** A'' **else** A''' **fi**$; A')\rangle]\!]\,\sigma =$
$$\begin{cases} \langle\sigma\rangle^\frown \mathscr{Comp}[\![\langle S, (A''; A')\rangle]\!]\,\sigma, & \text{if } \mathscr{W}[\![b]\!]\,\sigma = tt \\ \langle\sigma\rangle^\frown \mathscr{Comp}[\![\langle S, A'''; A')\rangle]\!]\,\sigma, & \text{if } \mathscr{W}[\![b]\!]\,\sigma = ff \end{cases}$$
7. $\mathscr{Comp}[\![\langle S,($**goto**$L; A')>]\!]\,\sigma = \langle\sigma\rangle^\frown \mathscr{Comp}[\![\langle S,$ **goto** $L\rangle]\!]\,\sigma$.

Some remarks on this definition will be useful. This style of defining is taken from Cook [9]. The definition should be viewed as a method for step-wise generating computation sequences. Each step will consist of replacing an occurrence of some expression $\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma$ using a rule from the definition. The rule to be applied depends on the form of A and is in fact uniquely determined by A. It is possible that this process won't terminate. In that case an element τ of Σ^{ω} will be generated. However this τ is well defined in the sense that every member of it is precisely determined.

The difficulties that arise by allowing **goto** statements in the language are reflected in clauses 4 to 7 of the definition. The problem is that $\mathscr{Comp}[\![\langle S, (A_1; A_2)\rangle]\!]\sigma$ cannot be defined easily in terms of $\mathscr{Comp}[\![\langle S, A_1\rangle]\!]$ and $\mathscr{Comp}[\![\langle S, A_2\rangle]\!]$, because evaluation of A_1 may terminate through execution of a substatement which is a jump out of A_1. The solution given here is to decompose a statement $(A_1; A_2)$, using rule 5 or 6, as long as it remains unclear whether an assignment or a jump has to be executed first. When this has become known, rule 4 or 7 can be applied.

The extra states $\langle\sigma\rangle$ which are added in the right-hand sides of clauses 2, 5, 6 and 7 are strictly speaking superfluous. They are introduced in order to be able to use induction in the proof of lemma 5.2 in a more elegant way. Note however that the $\langle\sigma\rangle$ added in rule 3 is necessary, because we want $\mathscr{Comp}[\![\langle L$: **goto** L, **goto** $L\rangle]\!]\,\sigma$ to be equal to $\langle\sigma, \sigma, \ldots\rangle \in \Sigma^{\omega}$, not to $\langle\sigma\rangle \in \Sigma^{+}$.

Finally, from the definition it can be seen that the following holds: if all labels in A and S are declared in S, then $\mathscr{C}omp[\![\langle S, A\rangle]\!]\sigma$ is defined for all σ.

We close this chapter by defining the operational meaning $\mathcal{O}[\![S]\!]$ for each program S in $\mathscr{P}rog$. This meaning will be a state transformation, i.e. an element of $(\Sigma \to \Sigma)$. The state $\mathcal{O}[\![S]\!]\sigma$ is meant to be the last element of the computation sequence τ, generated by evaluation of S starting in state σ. More precisely:

Definition ($\mathcal{O}$). The function $\mathcal{O}$ has functionality

$$\mathcal{O}\colon \mathscr{P}rog \to \Sigma \xrightarrow[p]{} \Sigma$$

and is defined by

$$\mathcal{O}[\![S]\!]\sigma = \kappa(\mathscr{C}omp[\![\langle S, A_1; \ldots A_n\rangle]\!]\sigma),$$

if $S \equiv [L_i : A_i]_{i=1}^n$.

4. Denotational Semantics

We now give semantical definitions in the style of Scott & Strachey [15], with additions (due to Strachey & Wadsworth [17] among others) to accomodate the peculiarities that goto-statements entail. The mathematical concepts used in these definitions are summarized below, so that we will be able to refer to them later on. Furthermore it can serve as a very concise introduction for those who are not yet acquainted with it. More details can be found in [5] or [16] for instance.

1. A pair $\langle C, \sqsubseteq\rangle$ is called a *complete partial order* (or a *cpo*) iff C is a non-empty set and $\sqsubseteq$ a partial order (i.e. a relation that is reflexive, transitive and anti-symmetric) such that

a) C contains a smallest element, called *bottom* and written as $\perp_C$ or just $\perp$, i.e. $\forall c \in C\colon \perp \sqsubseteq c$

b) Every sequence $c_1 \sqsubseteq c_2 \sqsubseteq \ldots$ of elements from C (called a *chain*, notation $\langle c_i\rangle_{i=1}^{\infty}$ or $\langle c_i\rangle_i$) has a least upper bound $\bigsqcup_i c_i$, satisfying

α) $\forall c_i\colon c_i \sqsubseteq \bigsqcup_j c_j$ (upper bound)

β) $\forall d \in C\colon [(\forall c_i\colon c_i \sqsubseteq d) \Rightarrow \bigsqcup_i c_i \sqsubseteq d]$ (the least one).

2. Σ as defined in the previous chapter, supplied with partial order $\sqsubseteq$, defined by

$$\sigma \sqsubseteq \sigma' \Leftrightarrow (\sigma = \sigma' \vee \sigma = \perp)$$

is a cpo. *A* cpo with partial order defined this way is called *discrete*.

3. Let A and B be cpo's, and $f \in (A \to B)$

a) f is called *monotonic* iff $\forall a, b \in A\colon a \sqsubseteq b \Rightarrow fa \sqsubseteq fb$

b) f is called *strict* iff $f\perp = \perp$. The class of all strict functions from A to B will be denoted by $(A \xrightarrow[s]{} B)$

c) f is called *continuous* iff f is monotonic and for every chain $\langle a_i \rangle_i$ in A, we have $f(\bigsqcup_i a_i) = \bigsqcup_i (fa_i)$. The class of all continuous functions in $(A \to B)$ will be denoted by $[A \to B]$.

4. Let A and B be cpo's. Then $[A \to B]$ is a cpo, if order, bottom and lub are defined by

a) $f \sqsubseteq g \Leftrightarrow \forall a \in A : fa \sqsubseteq ga$

b) $\bot_{[A \to B]} = \lambda a \cdot \bot_B$

c) if $\langle f_i \rangle_i$ is a chain then $\bigsqcup_i f_i = \lambda a \cdot \bigsqcup_i (f_i a)$.

5. Let A_i be a cpo for $i=1,\ldots,n$. Then $A_1 \times \ldots \times A_n$ is a cpo, if order, bottom and lub are defined by

a) $\langle a_1,\ldots,a_n \rangle \sqsubseteq \langle a'_1,\ldots,a'_n \rangle$ iff $a_i \sqsubseteq a'_i$ for $i=1,\ldots,n$

b) $\bot_{A_1 \times \ldots \times A_n} = \langle \bot,\ldots,\bot \rangle$

c) if $\langle a_1^{(i)},\ldots,a_n^{(i)} \rangle$ is a chain, then $\bigsqcup_i (\langle a_1^{(i)},\ldots,a_n^{(i)} \rangle) = \langle \bigsqcup_i a_1^{(i)},\ldots,\bigsqcup_i a_n^{(i)} \rangle$.

6. Let A be a cpo. Every continuous function $f \in [A \to A]$ has a *least fixed point*, written μf, with properties

a) $f(\mu f) = \mu f$ *fixed point property*, notation *fpp*

b) $\forall x \in A [f(x) \sqsubseteq x \Rightarrow \mu f \sqsubseteq x]$ *least fixed point property* (*lfp*)

c) $\mu f = \bigsqcup_i f^i(\bot)$, with $f^i(\bot)$ defined by $f^0(\bot) = \bot$, $f^{i+1}(\bot) = f(f^i(\bot))$.

We now discuss the denotational semantics for statements from $\mathscr{Stat}$. Again we are faced with problems about what to do with substatements of the form **goto** L. In the operational semantics this was solved by giving $\mathscr{Comp}$ an extra argument $S \equiv [L_i : A_i]_{i=1}^n$, which was used in essence to associate with each label L_i the statement $A_i; \ldots; A_n$. The meaning of **goto** L_i was practically the same as the maning of $A_i; \ldots; A_n$, which could be reduced to a state transformation (i.e. $\kappa \circ \mathscr{Comp}\, [\![\langle S, A_i; \ldots; A_n \rangle]\!]$, always a strict function). This function, applied to a state σ yields a final state σ', which is the result of evaluation of the statement $A_i; \ldots; A_n$. In other words, σ' is the result of evaluation of the rest of the program which will be executed after **goto** L_i has been evaluated.

The denotational semantics uses the same approach but in a more abstract way. Instead of giving for each label a program text that specificies a state transformation, we now provide this transformation directly. This is organized as follows: the semantic function $\mathscr{N}$ is given an extra argument γ, called an *environment*, which is a function from $\mathscr{Lvar}$ to $(\Sigma \xrightarrow[s]{} \Sigma)$. In the definition of $\mathscr{N}$ we then will have a clause like $\mathscr{N}[\![\textbf{goto}\ L]\!]\gamma = \gamma[\![L]\!]$. (How this $\gamma[\![L]\!]$ is obtained from the declaration of L in a program S will be discussed later when we come to define the meaning of programs.)

Thus we see that the meaning $\gamma[\![L]\!]$ of the statement **goto** L in an environment γ is a state transformation that doesn't describe the evaluation of **goto** L only, but also of the rest of the program to be evaluated once **goto** L has been executed. But then the same must be true for an arbitrary statement A as well. In the operational semantics care was taken of this, because the text of the rest of the program to be evaluated remained available (see clauses 4–6 in the definition of $\mathscr{C}omp$). Here we will use an abstraction of this idea resembling the approach of the **goto** statement. Instead of keeping track of a text defining a state transformation, we supply this transformation as an extra argument of $\mathscr{N}$. Such a transformation ϕ is called a *continuation*, and it is meant to describe the effect of evaluation of the "rest of the program", textually following the statement being defined. Summarizing: if ϕ specifies how evaluation of the program proceeds once the right-hand end of A has been reached, and if γ specifies for every label L how evaluation of the program proceeds once we have reached L, then we want $\mathscr{N}[\![A]\!]\gamma\phi$ to specify the evaluation of the program starting from the left-hand end of A.

This approach also solves the problem how to define the meaning of $(A_1; A_2)$ in terms of the meanings of A_1 and A_2. The meaning $\mathscr{N}[\![(A_1; A_2)]\!]\gamma\phi$ of $(A_1; A_2)$ in environment γ with continuation ϕ, will be equal to the meaning of A_1 in environment γ, but now with a new continuation ϕ'. For if evaluation of A_1 terminates normally (i.e. not through execution of a **goto** statement), then afterwards the statement A_2 has to be evalulated. Thus the continuation ϕ' must be equal to the meaning of A_2 in environment γ with continuation ϕ.

The exact definition of $\mathscr{N}$ will use some new domains which will be defined now:

a) $M = (\Sigma \xrightarrow[s]{} \Sigma)$ with typical elements ϕ, ψ, is the domain of the *continuations*. We use the convention that continuations appearing as an argument of $\mathscr{N}$ will be enclosed in curly brackets if that improves readability.

b) $\Gamma = (\mathscr{L}var \to M)$ with typical element γ, is the domain of the *environments*. We define a *variant* $\gamma\{\phi/L\}$ *of an environment* the same way as we did in the case of states: $(\gamma\{\phi/L\})[\![L']\!] = \gamma[\![L]\!]$ if $L' \not\equiv L$, and ϕ if $L' \equiv L$. We also use a simultaneous version: $(\gamma\{\phi_i/L_i\}_{i=1}^n)[\![L]\!] = \gamma[\![L]\!]$ if $L \not\equiv L_i$ for $i = 1, \ldots, n$, and ϕ_i if $L \equiv L_i$ (when we use such a construct, all L_i will be different).

Definition ($\mathscr{N}$). The semantic function $\mathscr{N}$ with functionality

$$\mathscr{N}: \mathscr{S}tat \to \Gamma \to M \to M$$

is defined inductively by

$$\mathscr{N}[\![x := s]\!]\gamma\phi\sigma = \begin{cases} \perp, & \text{if } \sigma = \perp \\ \phi(\sigma\{\mathscr{V}[\![s]\!]\sigma/x\}), & \text{otherwise,} \end{cases}$$

$$\mathscr{N}[\![(A_1; A_2)]\!]\gamma\phi\sigma = \mathscr{N}[\![A_1]\!]\gamma\{\mathscr{N}[\![A_2]\!]\gamma\phi\}\sigma,$$

$$\mathscr{N}[\![\textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}]\!]\gamma\phi\sigma = \begin{cases} \perp, & \text{if } \sigma = \perp \\ \mathscr{N}[\![A_1]\!]\gamma\phi\sigma, & \text{if } \sigma \neq \perp \text{ and } \mathscr{W}[\![b]\!]\sigma = tt \\ \mathscr{N}[\![A_2]\!]\gamma\phi\sigma, & \text{if } \sigma \neq \perp \text{ and } \mathscr{W}[\![b]\!]\sigma = ff, \end{cases}$$

$$\mathscr{N}[\![\textbf{goto } L]\!]\gamma\phi\sigma = \gamma[\![L]\!]\sigma.$$

The claim on the functionality of $\mathscr{N}$ in the above definition must be justified. It is though easy to show that $\forall A \in \mathscr{S}tat\ \forall\gamma \in \Gamma\ \forall\phi \in M$: $\mathscr{N}[\![A]\!]\gamma\phi \in M$.

The following lemma holds:

Lemma 4.1. *For all* $A \in \mathscr{S}tat$, *and all* $\gamma \in \Gamma$ *we have*

$$\lambda\langle\phi_1, \ldots, \phi_{n+1}\rangle \cdot \mathscr{N}[\![A]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\phi_{n+1} \in [M^{n+1} \to M].$$

Proof. Straightforward by induction on the structure of A. □

We now turn to the definition of the meaning of programs. The semantic function $\mathcal{M}: \mathscr{Prog} \to \Gamma \to M$ will be used for this purpose.

A program $S \equiv [L_i: A_i]_{i=1}^n$ can be considered as a combination of a statement $A_1; \ldots; A_n$ and a definition of the labels $L_1, \ldots, L_n$. The state $\mathcal{M}[\![[L_i: A_i]_{i=1}^n]\!] \gamma\sigma$ is meant to be the final state reached by evaluation of $A_1; \ldots; A_n$ starting in initial state σ, where the labels L_i are defined by S (for $i = 1, \ldots, n$) and all other labels by γ.

Definition ($\mathcal{M}$). $\mathcal{M}[\![[L_i: A_i]_{i=1}^n]\!] \gamma = \mathcal{N}[\![A_1; \ldots; A_n]\!] (\gamma\{\phi_i/L_i\}_{i=1}^n)\{\lambda\sigma \cdot \sigma\}$, where

$$\langle \phi_1, \ldots, \phi_n \rangle = \mu[\lambda \langle \psi_1, \ldots, \psi_n \rangle \cdot \langle \mathcal{N}[\![A_i]\!](\gamma\{\psi_j/L_j\}_{j=1}^n)\psi_{i+1}\rangle_{i=1}^n],$$
$$\text{where } \psi_{n+1} = \lambda\sigma \cdot \sigma.$$

Remarks. a) There is an assumption in the above definition that has to be justified. We have to show that the operator of which $\langle \phi_1, \ldots, \phi_n \rangle$ should be the least fixed point is a continuous one, i.e. a member of $[M^n \to M^n]$. In that case this least fixed point exists. The fact that this transformation is continuous can be proved using Lemma 4.1.

b) The function ϕ_i can intuitively be seen as the state transformation defined by evaluation of $A_i; \ldots; A_n$ where the labels are defined by S and γ. This might be clarified as follows. By *fpp* we have

$$\phi_n = \mathcal{N}[\![A_n]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\{\lambda\sigma \cdot \sigma\}$$

and

$$\begin{aligned}\phi_{n-1} &= \mathcal{N}[\![A_{n-1}]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\phi_n \\ &= \mathcal{N}[\![A_{n-1}]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\{\mathcal{N}[\![A_n]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\{\lambda\sigma \cdot \sigma\}\} \\ &= \mathcal{N}[\![A_{n-1}; A_n]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\{\lambda\sigma \cdot \sigma\}.\end{aligned}$$

Repeating this argument we get

$$\phi_i = \mathcal{N}[\![A_i; \ldots; A_n]\!](\gamma\{\phi_j/L_j\}_{j=1}^n)\{\lambda\sigma \cdot \sigma\} \qquad (i = 1, \ldots, n).$$

Moreover, these ϕ_i are precisely the values which we would expect to be associated with the labels L_i.

For later reference we state the following definitions and results.

Lemma 4.2. *Let $S \equiv [L_i: A_i]_{i=1}^n \in \mathscr{Prog}$, let $\gamma \in \Gamma$ and let ϕ_i be derived from S and γ as in the definition of $\mathcal{M}$. Also, let $\phi_i^{(k)}$ and $\gamma^{(k)}$ be defined inductively by:*

$$\begin{array}{ll} \phi_i^{(0)} = \lambda\sigma \cdot \bot & \textit{for } i = 1, \ldots, n \\ \phi_{n+1}^{(k)} = \lambda\sigma \cdot \sigma & \textit{for } k = 0, 1, \ldots \\ \gamma^{(k)} = \gamma\{\phi_j^{(k)}/L_j\}_{j=1}^n & \textit{for } k = 0, 1, \ldots \\ \phi_i^{(k+1)} = \mathcal{N}[\![A_i]\!]\gamma^{(k)}\phi_{i+1}^{(k)} & \textit{for } i = 1, \ldots, n \end{array}$$

4.2.1. $\phi_i = \bigsqcup_k \phi_i^{(k)} \qquad (i = 1, \ldots, n).$

Proof. This is a straightforward consequence of facts 5 and 6c from the theoretical remarks in the beginning of this section. □

4.2.2 $\phi_i = \mathcal{N}[\![A_i; \ldots; A_n]\!](\gamma\{\phi_j/L_j\}_{j=1}^n)\{\lambda\sigma\cdot\sigma\}$.

Proof. See remark b) above. □

4.2.3. $\phi_i^{(k)} \sqsubseteq \mathcal{N}[\![A_i; \ \ldots; A_n]\!]\gamma^{(k-1)}\{\lambda\sigma\cdot\sigma\} \qquad (1 \leqq i \leqq n,\ k = 1, 2, \ldots)$.

Proof. Induction on k. The basic step ($k=1$) can be proved using the fact that $\lambda\phi\cdot\mathcal{N}[\![A]\!]\gamma\phi$ is monotonic and that

$$\mathcal{N}[\![A_i; \ldots A_n]\!]\gamma\phi = \mathcal{N}[\![A_i]\!]\gamma\{\mathcal{N}[\![A_{i+1}; \ldots; A_n]\!]\gamma\phi\}.$$

The induction step is proved as follows:

$$\begin{aligned}\phi_i^{(k)} &= \mathcal{N}[\![A_i]\!]\gamma^{(k-1)}\phi_{i+1}^{(k-1)} \\ &= \mathcal{N}[\![A_i]\!]\gamma^{(k-1)}\{\mathcal{N}[\![A_{i+1}]\!]\gamma^{(k-2)}\phi_{i+2}^{(k-2)}\} = (\#).\end{aligned}$$

Now we use Lemma 4.1, and the fact that continuity implies monotonicity to show that $\mathcal{N}[\![A_{i+1}]\!]\gamma^{(k-2)}\phi_{i+2}^{(k-2)} \sqsubseteq \mathcal{N}[\![A_{i+1}]\!]\gamma^{(k-1)}\phi_{i+2}^{(k-1)}$ and thus, using 4.1 again:

$$\begin{aligned}(\#) &\sqsubseteq \mathcal{N}[\![A_i]\!]\gamma^{(k-1)}\{\mathcal{N}[\![A_{i+1}]\!]\gamma^{(k-1)}\phi_{i+2}^{(k-1)}\} \\ &= \mathcal{N}[\![A_i; A_{i+1}]\!]\gamma^{(k-1)}\phi_{i+2}^{(k-1)}.\end{aligned}$$

Repeating the argument we get

$$\begin{aligned}\phi_i^{(k)} &\sqsubseteq \mathcal{N}[\![(\ldots(A_i; A_{i+1}); \ldots); A_n)]\!]\gamma^{(k-1)}\{\lambda\sigma\cdot\sigma\} \\ &= \mathcal{N}[\![A_i; A_{i+1}; \ldots; A_n]\!]\gamma^{(k-1)}\{\lambda\sigma\cdot\sigma\},\end{aligned}$$

where the last identity is easy to prove from the definition of $\mathcal{N}$. □

We will close this section by taking another look at the meaning of statements A. We saw that the function $\mathcal{N}[\![A]\!]$ essentially yields a continuation as a result. This result depends on a number of continuations, which are supplied to $\mathcal{N}[\![A]\!]$ either directly as an argument (the ϕ in $\mathcal{N}[\![A]\!]\gamma\phi$) or implicitly through γ, as meaning of the labels occurring in A. In the literature [13] a method called *continuation removal* is described to dispose of the ϕ in the above formula, yielding a more direct approach: the meaning of a statement is a state transformation instead of a continuation transformation. This has only been done for statements A which didn't contain **goto** statements as substatements.

We now take one further step: we show how to deal with **goto**-substatements. We will define a function $\mathcal{A}$ giving the meaning of a statement A as a (total) function from Σ_0 to $\Sigma_0 \cup (\Sigma_0 \times \mathcal{L}var)$, such that

$$\mathcal{A}[A]\sigma = \sigma'$$

means that evaluation of A terminates normally in state σ' (i.e. not as the result of an execution of a **goto** statement), and

$$\mathcal{A}[\![A]\!]\sigma = \langle\sigma', L\rangle$$

means that evaluation of A terminates by execution of a substatement **goto** L in state σ'.

Put another way, a statement A containing **goto**-substatements can be viewed as a statement with one *entry point* (where evaluation of A starts), but with several *exit points*, namely the *normal exit point* (the right-hand end of the statement) and the *special exit points* (viz. the substatements **goto** L). We call an exit point determined by a substatement **goto** L an *L-exit point*. The function $\mathscr{A}$ then specifies for every initial state σ the kind of exit point which will be reached and the final state in which this exit point will be reached. This is a formalization of the considerations in [3].

The function $\mathscr{A}$, applied to a statement A and an initial state σ, thus yields a final state σ' which is the result of evaluation of A, and not of evaluation of A followed by some continuation (as was the case in $\mathscr{N}[\![A]\!]\gamma\phi\sigma$). Since in the deduction systems to be discussed later we deal with formulae $\{p\}\,A\{q\}$, where q is a predicate on the final state at the normal exit point, we can expect that the function $\mathscr{A}$ will be more useful than $\mathscr{N}$ (see Sect. 6).

We now give the definition of $\mathscr{A}$.

Definition ($\mathscr{A}$). The function $\mathscr{A}$ with functionality $\mathscr{A}\colon \mathscr{S}tat \to \Sigma_0 \to \Sigma_0 \cup (\Sigma_0 \times \mathscr{L}var)$ is inductively defined by

$$\mathscr{A}[\![x:=s]\!]\,\sigma = \sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\}$$

$$\mathscr{A}[\![(A_1;A_2)]\!]\,\sigma = \begin{cases} \mathscr{A}[\![A_2]\!](\mathscr{A}[\![A_1]\!]\,\sigma), & \text{if } \mathscr{A}[\![A_1]\!]\,\sigma \in \Sigma_0 \\ \mathscr{A}[\![A_1]\!]\,\sigma, & \text{otherwise} \end{cases}$$

$$\mathscr{A}[\![\textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}]\!]\,\sigma = \begin{cases} \mathscr{A}[\![A_1]\!]\,\sigma, & \text{if } \mathscr{W}[\![b]\!]\,\sigma = tt \\ \mathscr{A}[\![A_2]\!]\,\sigma, & \text{if } \mathscr{W}[\![b]\!]\,\sigma = ff \end{cases}$$

$$\mathscr{A}[\![\textbf{goto } L]\!]\,\sigma = \langle \sigma, L\rangle.$$

We have the following lemma on the relation between $\mathscr{A}$ and $\mathscr{N}$.

Lemma 4.3. 1°. $\mathscr{A}[\![A]\!]\,\sigma = \sigma' \Leftrightarrow \forall\gamma\in\Gamma\ \forall\phi\in M\colon \mathscr{N}[\![A]\!]\,\gamma\phi\sigma = \phi\sigma'$

2°. $\mathscr{A}[\![A]\!]\,\sigma = \langle\sigma', L\rangle \Leftrightarrow \forall\gamma\in\Gamma\ \forall\phi\in M\colon \mathscr{N}[\![A]\!]\,\gamma\phi\sigma = \gamma[\![L]\!]\,\sigma'$.

Proof. The $\Rightarrow$-parts of 1° and 2° are straightforward by structural induction. The $\Leftarrow$-parts can be proven by contradiction. For instance, proving 2° "$\Leftarrow$", suppose $\forall\gamma\in\Gamma\ \forall\phi\in M\colon \mathscr{N}[\![A]\!]\,\gamma\phi\sigma = \gamma[\![L]\!]\,\sigma'$, and $\mathscr{A}[\![A]\!]\,\sigma \neq \langle\sigma', L\rangle$. Then we have two possibilities.

The first one is $\mathscr{A}[\![A]\!]\,\sigma = \langle\sigma'', L'\rangle$ (where $\sigma' \neq \sigma''$ or $L \neq L'$) and thus, using 2° "$\Rightarrow$", $\forall\gamma\in\Gamma\ \forall\phi\in M\colon \mathscr{N}[\![A]\!]\,\gamma\phi\sigma = \gamma[\![L']\!]\,\sigma''$. Now choose γ such that $\gamma[\![L]\!]\,\sigma' \neq \gamma[\![L']\!]\,\sigma''$ and we have a contradiction.

The other possibility is $\mathscr{A}[\![A]\!]\sigma = \sigma''$. Then we have (1° "$\Rightarrow$") $\forall\gamma\in\Gamma\ \forall\phi\in M\colon \mathscr{N}[\![A]\!]\,\gamma\phi\sigma = \phi\sigma''$, and we reach a contradiction by choosing γ and ϕ such that $\gamma[\![L]\!]\,\sigma' \neq \phi\sigma''$. □

5. Operational and Denotational Semantics are Equivalent

Our aim in this section is to prove the following

Theorem 5.1. *Let $S \equiv [L_i : A_i]_{i=1}^n$ be a program. If S is normal (i.e. all labels in S are declared) then*

$$\forall\gamma\in\Gamma\colon \mathscr{O}[\![S]\!] = \mathscr{M}[\![S]\!]\,\gamma.$$

Proof. We first prove $\mathscr{O}[\![S]\!] \sqsubseteq \mathscr{M}[\![S]\!]\,\gamma$, using the following

Lemma 5.2. *Let $S \equiv [L_i : A_i]_{i=1}^n \in \mathscr{Prog}$, $\gamma \in \Gamma$, and let ϕ_i be derived from S and γ as in the definition of $\mathscr{M}$. Let $A \in \mathscr{Stat}$ and let all labels occurring in A and S be declared in S. Then*

$$\forall \sigma \in \Sigma : \kappa(\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma) \sqsubseteq \mathscr{N}[\![A]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\{\lambda\sigma\cdot\sigma\}\,\sigma \qquad \ldots(*)$$

Proof of the Lemma. Because all labels are declared in S, it is impossible that $\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma$ be undefined. If $\sigma = \bot$, or if $\sigma \neq \bot$ and $\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma \in \Sigma^\omega$, then the left-hand side of $(*)$ is equal to $\bot$, and the inequality holds. So let us assume that $\sigma \neq \bot$ and $\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma \in \Sigma^+$. We prove $(*)$ using induction on the length of the computation sequence $\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma$ (in fact, assuming that this computation sequence is finite, we can prove equality in $(*)$). We distinguish several cases, depending on the structure of A. We shall abbreviate $\gamma\{\phi_i/L_i\}_{i=1}^n$ to $\bar{\gamma}$.

a) $A \equiv x := s$. Then the left-hand side of $(*)$ is equal to $\sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\}$, and so is the right-hand side.

b) $A \equiv$ **if** b **then** A_1 **else** A_2 **fi**. Assume $\mathscr{W}[\![b]\!]\,\sigma = tt$ (the other case can be proved analogously). Then $\kappa(\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma) = \kappa(\langle\sigma\rangle^\frown \mathscr{Comp}[\![\langle S, A_1\rangle]\!]\,\sigma)$. Now the length of $\mathscr{Comp}[\![\langle S, A_1\rangle]\!]\,\sigma$ is clearly one less than the length of $\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma$, so we can apply the induction hypothesis, yielding

$$\kappa(\mathrm{Comp}[\![\langle S, A\rangle]\!]\,\sigma) = \kappa(\mathscr{Comp}[\![\langle S, A_1\rangle]\!]\,\sigma)$$
$$\sqsubseteq \mathscr{N}[\![A_1]\!]\,\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\,\sigma = \mathscr{N}[\![A]\!]\,\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\,\sigma.$$

c) $A \equiv$ **goto** L. Because all labels in A are defined in S, we have $L \equiv L_j$ for some j. Thus

$$\begin{aligned}
&\kappa(\mathscr{Comp}[\![\langle [L_i : A_i]_{i=1}^n, A\rangle]\!]\,\sigma) \\
&= \kappa(\langle\sigma\rangle^\frown \mathscr{Comp}[\![\langle S, A_j; \ldots; A_n\rangle]\!]\,\sigma) && \text{(ind. hyp.)} \\
&\sqsubseteq \mathscr{N}[\![A_j; \ldots; A_n]\!]\,\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\,\sigma && (4.2.2) \\
&= \phi_j\sigma = \bar{\gamma}[\![L_j]\!]\,\sigma \\
&= \mathscr{N}[\![\textbf{goto } L_j]\!]\,\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\,\sigma.
\end{aligned}$$

d) $A \equiv (x := s; A')$.

$$\begin{aligned}
\kappa(\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma) &= \kappa(\mathscr{Comp}[\![\langle S, A'\rangle]\!]\,(\sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\})) && \text{(ind.)} \\
&\sqsubseteq \mathscr{N}[\![A']\!]\,\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\,(\sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\}) && \text{(def. } \mathscr{N}) \\
&= \mathscr{N}[\![x := s]\!]\,\bar{\gamma}\{\mathscr{N}[\![A']\!]\,\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\}\,\sigma \\
&= \mathscr{N}[\![(x := s; A')]\!]\,\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\,\sigma.
\end{aligned}$$

e) $A \equiv ((A_1; A_2); A')$

$$\begin{aligned}
\kappa(\mathscr{Comp}[\![\langle S, A\rangle]\!]\,\sigma) &= \kappa(\mathscr{Comp}[\![\langle S, (A_1; (A_2; A'))\rangle]\!]\,\sigma) && \text{(ind.)} \\
&\sqsubseteq \mathscr{N}[\![(A_1; (A_2; A'))]\!]\,\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\,\sigma \\
&= \mathscr{N}[\![((A_1; A_2); A')]\!]\,\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\,\sigma.
\end{aligned}$$

f) $A \equiv (\textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}; A')$. We suppose, without loss of generality, that $\mathscr{W}[\![b]\!]\sigma = tt$. Then

$$\begin{aligned}
\kappa(\mathscr{C}omp[\![\langle S, A\rangle]\!]\sigma) &= \kappa(\mathscr{C}omp[\![\langle S, A_1; A'\rangle]\!]\sigma) && \text{(ind. hyp.)}\\
&\sqsubseteq \mathscr{N}[\![A_1; A']\!]\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\sigma && \text{(def. } \mathscr{N})\\
&= \mathscr{N}[\![A_1]\!]\bar{\gamma}\{\mathscr{N}[\![A']\!]\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\}\sigma && \text{(def. } \mathscr{N})\\
&= \mathscr{N}[\![\textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}]\!]\bar{\gamma}\{\mathscr{N}[\![A']\!]\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\}\sigma\\
&= \mathscr{N}[\![\textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}; A')]\!]\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\sigma.
\end{aligned}$$

g) $A \equiv (\textbf{goto } L; A')$.

$$\begin{aligned}
\kappa(\mathscr{C}omp[\![\langle S, A\rangle]\!]\sigma) &= \kappa(\mathscr{C}omp[\![\langle S, \textbf{goto } L\rangle]\!]\sigma) && \text{(ind. hyp.)}\\
&\sqsubseteq \mathscr{N}[\![\textbf{goto } L]\!]\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\sigma && \text{(def. } \mathscr{N})\\
&= \mathscr{N}[\![\textbf{goto } L]\!]\bar{\gamma}\{\mathscr{N}[\![A']\!]\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\}\sigma && \text{(def. } \mathscr{N})\\
&= \mathscr{N}[\![(\textbf{goto } L; A')]\!]\bar{\gamma}\{\lambda\sigma\cdot\sigma\}\sigma.
\end{aligned}$$

This ends the proof of Lemma 5.2. □

We now use the lemma to prove $\mathscr{O}[\![s]\!] \sqsubseteq \mathscr{M}[\![S]\!]\gamma$ in the following way. Choose $\gamma \in \Gamma$ and $\sigma \in \Sigma$. By definition of $\mathscr{O}$ we have

$$\mathscr{O}[\![S]\!]\sigma = \kappa(\mathscr{C}omp[\![\langle S, A_1; \ldots; A_n\rangle]\!]\sigma).$$

Now all labels in S are declared and thus the same holds for $A_1; \ldots; A_n$. The lemma then gives us

$$\kappa(\mathscr{C}omp[\![\langle S, A_1; \ldots; A_n\rangle]\!]\sigma) \sqsubseteq \mathscr{N}[\![A_1; \ldots; A_n]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\{\lambda\sigma\cdot\sigma\}\sigma,$$

where the ϕ_i are obtained from S and γ as in the definition of $\mathscr{M}$. But, for those ϕ_i, the definition of $\mathscr{M}$ gives us

$$\mathscr{N}[\![A_1; \ldots; A_n]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\{\lambda\sigma\cdot\sigma\}\sigma = \mathscr{M}[\![S]\!]\gamma\sigma$$

which gives us the desired result.

For the proof of $\mathscr{M}[\![S]\!]\gamma \sqsubseteq \mathscr{O}[\![S]\!]$, we again use a lemma:

Lemma 5.3. *Let $S \equiv [L_i : A_i]_{i=1}^n \in \mathscr{P}rog$ be normal. Let $\gamma \in \Gamma$ and $k \in \mathbb{N}$. Let ϕ_i, $\phi_i^{(k)}$ and $\gamma^{(k)}$ be derived from S and γ as in lemma 4.2. Then, for all $A \in \mathscr{S}tat$ such that all labels in A are declared in S, and for all $\sigma \in \Sigma$, we have*

$$\mathscr{N}[\![A]\!]\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}\sigma \sqsubseteq \kappa(\mathscr{C}omp[\![\langle S, A\rangle]\!]\sigma) \qquad \ldots(+)$$

Proof of the Lemma. We use induction on the entity $\langle k, c[\![A]\!]\rangle$ with lexicographic ordering $\prec$. We don't take $c[\![A]\!]$ to be the obvious complexity of A. This wouldn't work because of the form of the definition of $\mathscr{C}omp$. For instance, in rule 5 the statement $(A''; (A'''; A'))$ occurring in the right-hand side of the rule would be as complex, according to the usual complexity measure, as $((A''; A'''); A')$ in the left-hand side.

We define $c[\![A]\!]$ inductively by: $c[\![x:=s]\!]=c[\![\textbf{goto } L]\!]=1$; $c[\![A_1; A_2]\!]=2c[\![A_1]\!]+c[\![A_2]\!]$; $c[\![\textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}]\!]=c[\![A_1]\!]+c[\![A_2]\!]$.

a) $A \equiv x:=s$. Then

$$\mathcal{N}[\![x:=s]\!]\,\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}\,\sigma=\kappa(\mathcal{C}omp[\![\langle S, A_1\rangle]\!]\,\sigma)=\sigma\{\mathcal{V}[\![s]\!]\,\sigma/x\}.$$

b) $A \equiv \textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}$. Without loss of generality, we assume $\mathcal{W}[\![b]\!]\,\sigma=tt$. Then the left-hand side of $(+)$ equals $\mathcal{N}[\![A_1]\!]\,\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}\sigma$, and the right-hand side equals $\kappa(\mathcal{C}omp[\![\langle S, A_1\rangle]\!]\,\sigma)$. Now, because $\langle k, c[\![A_1]\!]\rangle \prec \langle k, c[\![A]\!]\rangle$, the desired result follows from the induction hypothesis.

c) $A \equiv \textbf{goto } L$. From the assumptions of the lemma, we infer that $L \equiv L_i$ for some i. Now:

$$\begin{aligned}\mathcal{N}[\![\textbf{goto } L_i]\!]\,\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}\,\sigma&=(\gamma^{(k)})[\![L_i]\!]\,\sigma=\phi_i^{(k)}\sigma\\&\sqsubseteq\mathcal{N}[\![A_i; \ldots; A_n]\!]\,\gamma^{(k-1)}\{\lambda\sigma\cdot\sigma\}\,\sigma. \qquad (4.2.3)\end{aligned}$$

Also $\langle k-1, c[\![A_i; \ldots; A_n]\!]\rangle \prec \langle k, c[\![\textbf{goto } L_i]\!]\rangle$, so we can apply the induction hypothesis

$$\begin{aligned}&\mathcal{N}[\![A_i; \ldots; A_n]\!]\,\gamma^{(k-1)}\{\lambda\sigma\cdot\sigma\}\,\sigma\\&\quad\sqsubseteq\kappa(\mathcal{C}omp[\![\langle S, A_i; \ldots; A_n\rangle]\!]\,\sigma) \qquad (\text{def. } \mathcal{C}omp)\\&\quad=\kappa(\mathcal{C}omp[\![\langle S, \textbf{goto } L_i\rangle]\!]\,\sigma).\end{aligned}$$

d) $A \equiv (x:=s; A')$.

$$\begin{aligned}&\mathcal{N}[\![(x:=s; A')]\!]\,\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}\,\sigma\\&\quad=\mathcal{N}[\![A']\!]\,\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}(\sigma\{\mathcal{V}[\![s]\!]\,\sigma/x\}) \qquad (c[\![A']\!]\prec c[\![x:=s; A']\!])\\&\quad\sqsubseteq\kappa(\mathcal{C}omp[\![\langle S, A'\rangle]\!]\,(\sigma\{\mathcal{V}[\![s]\!]\,\sigma/x\}))\\&\quad=\kappa(\mathcal{C}omp[\![\langle S, (x:=s; A')\rangle]\!]\,\sigma).\end{aligned}$$

e) $A \equiv ((A_1; A_2); A')$. We have $\mathcal{N}[\![((A_1; A_2); A')]\!]=\mathcal{N}[\![(A_1; (A_2; A'))]\!]$ and $c[\![(A_1; (A_2; A'))]\!] \prec c[\![((A_1; A_2); A')]\!]$. The induction hypothesis thus yields

$$\begin{aligned}&\mathcal{N}[\![(A_1; (A_2; A'))]\!]\,\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}\,\sigma\\&\quad\sqsubseteq\kappa(\mathcal{C}omp[\![\langle S, (A_1; (A_2; A'))\rangle]\!]\,\sigma)\\&\quad=\kappa(\mathcal{C}omp[\![\langle S, ((A_1; A_2); A')\rangle]\!]\,\sigma).\end{aligned}$$

f) $A \equiv (\textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}; A')$. Without loss of generality, we assume that $\mathcal{W}[\![b]\!]\,\sigma=tt$. We then have

$$\begin{aligned}\mathcal{N}[\![A]\!]\,\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}\,\sigma&=\mathcal{N}[\![A_1; A']\!]\,\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}\,\sigma \qquad (\text{ind. hyp.})\\&\sqsubseteq\kappa(\mathcal{C}omp[\![\langle S, A_1; A'\rangle]\!]\,\sigma)=\kappa(\mathcal{C}omp[\![\langle S, A\rangle]\!]\,\sigma).\end{aligned}$$

g) $A \equiv (\textbf{goto } L; A')$.

$$\begin{aligned}\mathcal{N}[\![(\textbf{goto } L; A')]\!]\,\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}\,\sigma&=\mathcal{N}[\![\textbf{goto } L]\!]\,\gamma^{(k)}\{\lambda\sigma\cdot\sigma\}\,\sigma \qquad (\text{ind. hyp.})\\&\sqsubseteq\kappa(\mathcal{C}omp[\![\langle S, \textbf{goto } L\rangle]\!]\,\sigma)=\kappa(\mathcal{C}omp[\![\langle S, \textbf{goto } L; A'\rangle]\!]\,\sigma).\end{aligned}$$

This concludes the proof of Lemma 5.3. □

We now are able to prove $\mathcal{M}[\![S]\!]\gamma \sqsubseteq \mathcal{O}[\![S]\!]$.

$$\begin{aligned}\mathcal{M}[\![S]\!]\gamma &= \mathcal{N}[\![A_1; \dots; A_n]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\{\lambda\sigma\cdot\sigma\}\\ &= \mathcal{N}[\![A_1; \dots; A_n]\!](\gamma\{\bigsqcup_k \phi_i^{(k)}/L_i\}_{i=1}^n)\{\lambda\sigma\cdot\sigma\} \qquad \text{(Lemma 4.1)}\\ &= \bigsqcup_k (\mathcal{N}[\![A_1; \dots; A_n]\!](\gamma\{\phi_i^{(k)}/L_i\}_{i=1}^n)\{\lambda\sigma\cdot\sigma\}).\end{aligned}$$

Now, taking $A \equiv A_1; \dots; A_n$, the assumptions of Lemma 5.3 are satisfied. Thus we can conclude

$$\forall k \in \mathbb{N}: \mathcal{N}[\![A_1; \dots; A_n]\!](\gamma\{\phi_i^{(k)}/L_i\}_{i=1}^n)\{\lambda\sigma\cdot\sigma\} \sqsubseteq \kappa \circ \mathcal{C}omp[\![\langle S, A_1; \dots; A_n\rangle]\!],$$

and thus

$$\bigsqcup_k \mathcal{N}[\![A_1; \dots; A_n]\!](\{\phi_i^{(k)}/L_i\}_{i=1}^n)\{\lambda\sigma\cdot\sigma\} \sqsubseteq \kappa \circ \mathcal{C}omp[\![\langle S, A_1; \dots; A_n\rangle]\!].$$

But also, by definition of $\mathcal{O}$:

$$\kappa \circ \mathcal{C}omp[\![\langle S, A_1; \dots; A_n\rangle]\!] = \mathcal{O}[\![S]\!],$$

which completes the proof. □

Note that Theorem 5.1 is independent of the interpretation of the primitive relation and function symbols chosen, in the sense that the theorem holds for all underlying interpretations $\mathcal{I}$.

6. Deduction System: First Variant

In [10] Hoare proposed to attach meanings to programs by means of a proof system which can be used to derive properties of programs. These properties are described by (partial) *correctness formulae*, essentially having the form $\{p\}\,S\,\{q\}$. Such a construct has informally the following meaning: if evaluation of S terminates, starting from an initial state in which p (the *precondition*) holds, then in the final state q (the *postcondition*) holds.

We start with a discussion of these conditions p, which in the sequel will be called *assertions*. The class of all assertions is $\mathcal{A}ssn$, with typical elements p, q. We define:

$$p ::= \textbf{true} \mid p_1 \vee p_2 \mid \neg p \mid re_1(s_1, \dots, s_{arr_1}) \mid \dots \mid re_n(s_1, \dots, s_{arr_n}) \mid \exists x[p].$$

We define **false**, $p_1 \wedge p_2$, $p_1 \supset p_2$ and **if** b **then** p_1 **else** p_2 **fi** as in Sect. 2.

The assertions are meant to describe predicates on states. The semantic function giving the meaning of assertions is $\mathcal{T}$ and has functionality

$$\mathcal{T}: \mathcal{A}ssn \to \Sigma_0 \to \{ff, tt\}.$$

$\mathcal{T}$ is defined inductively by:

a) $\mathcal{T}[\![\textbf{true}]\!]\sigma = tt$

b) $\mathcal{T}[\![p_1 \vee p_2]\!]\sigma = tt$ if $\mathcal{T}[\![p_1]\!]\sigma = tt$ or $\mathcal{T}[\![p_2]\!]\sigma = tt$, and ff otherwise

c) $\mathscr{T}[\![\neg p]\!]\sigma = tt$ if $\mathscr{T}[\![p]\!]\sigma = ff$, and ff otherwise

d) $\mathscr{T}[\![re_i(s_1, \ldots, s_{arr_i})]\!]\sigma = tt$ if $\langle \mathscr{V}[\![s_1]\!]\sigma, \ldots, \mathscr{V}[\![s_{arr_i}]\!]\sigma\rangle \in \underline{re}_i$, and ff otherwise

e) $\mathscr{T}[\![\exists x[p]]\!]\sigma = tt$ if there exists an element d in our domain of interpretation D such that $\mathscr{T}[\![p]\!](\sigma\{d/x\}) = tt$, and ff otherwise.

Note that $\mathscr{T}$ depends on the underlying interpretation $\mathscr{I}$, because $\mathscr{V}$ does (d)), and also through clause e) of the definition.

Next some definitions and results on substitution. We say that an occurrence of a variable x in an assertion p is *bound*, if this occurrence is within a sub-assertion of the form $\exists x[p']$. An occurrence of x in an assertion p is called *free* if it is not bound.

The result of substituting all (free) occurrences of x in s and p by t, will be denoted by $s[t/x]$ and $p[t/x]$ respectively. The definition of $s[t/x]$ is

a) $y[t/x] \equiv t$ if $y \equiv x$ and y otherwise

b) $(fu_i(s_1, \ldots, s_{arf_i}))[t/x] \equiv fu_i(s_1[t/x], \ldots, s_{arf_i}[t/x])$.

Using this definition $p[t/x]$ can be defined by

a) **true** $[t/x] \equiv$ **true**

b) $(p_1 \vee p_2)[t/x] \equiv p_1[t/x] \vee p_2[t/x]$

c) $(\neg p)[t/x] \equiv \neg(p[t/x])$

d) $(re_i(s_1, \ldots, s_{arr_i}))[t/x] \equiv re_i(s_1[t/x], \ldots, s_{arr_i}[t/x])$

e) $$(\exists y[p])[t/x] \equiv \begin{cases} \exists y[p], \text{ if } x \equiv y \\ \exists y[p[t/x]], \text{ if } x \not\equiv y \text{ and y does not occur in } t \\ \exists z[p[z/y][t/x]] \text{ if } x \not\equiv y \text{ and } y \text{ occurs in } t, \text{ where } z \\ \quad \text{is the first variable in } \mathscr{V}ar \text{ such that } z \not\equiv x, \\ \quad z \text{ doesn't occur in } t, z \text{ doesn't occur free in } p. \end{cases}$$

The following results on substitution will be useful.

Lemma 6.1. a) *If x doesn't occur in s then $\forall d \in D$: $\mathscr{V}[\![s]\!]\sigma = \mathscr{V}[\![s]\!](\sigma\{d/x\})$*

b) *if x doesn't occur free in p then $\forall d \in D$: $\mathscr{T}[\![p]\!]\sigma = \mathscr{T}[\![p]\!](\sigma\{d/x\})$*

c) $\mathscr{V}[\![s[t/x]]\!]\sigma = \mathscr{V}[\![s]\!](\sigma\{\mathscr{V}[\![t]\!]\sigma/x\})$

d) $\mathscr{T}[\![p[t/x]]\!]\sigma = \mathscr{T}[\![p]\!](\sigma\{\mathscr{V}[\![t]\!]\sigma/x\})$.

Proof. Straightforward by induction. We prove the hardest case of d), i.e. where the assertion has the form $\exists y[p]$. There are three cases.

1) $y \equiv x$. $\mathscr{T}[\![\exists x[p][t/x]]\!]\sigma = \mathscr{T}[\![\exists x[p]]\!]\sigma = tt$ iff $\exists d \in D$: $\mathscr{T}[\![p]\!](\sigma\{d/x\}) = tt$. Now $\mathscr{T}[\![p]\!](\sigma\{d/x\}) = \mathscr{T}[\![p]\!](\sigma\{\mathscr{V}[\![t]\!]\sigma/x\}\{d/x\})$, and therefore $\mathscr{T}[\![\exists x[p]]\!]\sigma = tt$, iff $\exists d \in D$: $\mathscr{T}[\![p]\!](\sigma\{\mathscr{V}[\![t]\!]\sigma/x\}\{d/x\}) = tt$, and this is true whenever $\mathscr{T}[\![\exists x[p]]\!](\sigma\{\mathscr{V}[\![t]\!]\sigma/x\}) = tt$.

2) $y \not\equiv x$ and y doesn't occur in t. $\mathscr{T}[\![(\exists y[p])[t/x]]\!]\sigma = \mathscr{T}[\![\exists y[p[t/x]]]\!]\sigma = tt$
iff $\exists d \in D$: $\mathscr{T}[\![p[t/x]]\!](\sigma\{d/y\}) = tt$ (ind. hyp.)
iff $\exists d \in D$: $\mathscr{T}[\![p]\!](\sigma\{d/y\}\{\mathscr{V}[\![t]\!](\sigma\{d/y\})/x\}) = tt$ (a))
iff $\exists d \in D$: $\mathscr{T}[\![p]\!](\sigma\{\mathscr{V}[\![t]\!]\sigma/x\}\{d/y\}) = tt$
iff $\mathscr{T}[\![\exists y[p]]\!](\sigma\{\mathscr{V}[\![t]\!]\sigma/x\}) = tt$.

3) $y \not\equiv x$ and y occurs in t. $\mathscr{T}[\![\exists y[p][t/x]]\!]\sigma = \mathscr{T}[\![\exists z[p[z/y][t/x]]]\!]\sigma = (\#)$ where $z \not\equiv x$, z doesn't occur in t, z doesn't occur free in p.

Now $(\#) = tt$ iff $\exists d \in D$: $\mathscr{T}[\![p[z/y][t/x]]\!](\sigma\{d/z\}) = tt$ (ind. hyp.)
iff $\exists d \in D$: $\mathscr{T}[\![p]\!](\sigma\{d/z\}\{\mathscr{V}[\![t]\!]\sigma'/x\}\{\sigma''[\![x]\!]/y\}) = tt$,

where $\sigma' = \sigma\{d/z\}$ and $\sigma'' = \sigma'\{\mathscr{V}[\![t]\!]\sigma'/x\}$. Now $x \not\equiv z$, so $\sigma''[\![z]\!] = d$. Furthermore z doesn't occur in t, so $\mathscr{V}[\![t]\!]\sigma' = \mathscr{V}[\![t]\!](\sigma\{d/z\}) = \mathscr{V}[\![t]\!]\sigma$. Thus we get

$$(\#) = tt \quad \text{iff } \exists d \in D\colon \mathscr{T}[\![p]\!](\sigma\{d/z\}\{\mathscr{V}[\![t]\!]\sigma/x\}\{d/y\}) = tt.$$

Because z doesn't occur in t, and y does, we have $z \not\equiv y$. Also we have $z \not\equiv x$ and $y \not\equiv x$, so

$$(\#) = tt \quad \text{iff } \exists d \in D\colon \mathscr{T}[\![p]\!](\sigma\{\mathscr{V}[\![t]\!]\sigma/x\}\{d/y\}\{d/z\}) = tt.$$

Because z doesn't occur free in p, we can use result b) of the lemma, to get

$$\begin{aligned}(\#) = tt \quad &\text{iff } \exists d \in D\colon \mathscr{T}[\![p]\!](\sigma\{\mathscr{V}[\![t]\!]\sigma/x\}\{d/y\}) = tt\\ &\text{iff } \mathscr{T}[\![\exists y[p]]\!](\sigma\{\mathscr{V}[\![t]\!]\sigma/x\}) = tt. \quad \square\end{aligned}$$

Having defined assertions and substitution, we now proceed to describe how these notions are to be used in correctness formulae. A typical axiom of our proof system will be the *assignment axiom*, roughly of the form

$$\{p[s/x]\}\, x := s\, \{p\}.$$

This axiom can be justified by the following considerations. The statement $x := s$ transforms an initial state σ to a final state $\sigma' = \sigma\{\mathscr{V}[\![s]\!]\sigma/x\}$. Now suppose $p[s/x]$ is true in σ, that is, $\mathscr{T}[\![p[s/x]]\!]\sigma = tt$, or (Lemma 6.1d) $\mathscr{T}[\![p]\!](\sigma\{\mathscr{V}[\![s]\!]\sigma/x\}) = tt$. But $\sigma\{\mathscr{V}[\![s]\!]\sigma/x\}$ is equal to the final state σ', so we have that p is true in σ', which is what we wanted.

A rule of inference in the system will be the *rule of composition*, stated informally

$$\text{from } \{p_1\}A_1\{p_2\} \text{ and } \{p_2\}A_2\{p_3\} \text{ infer } \{p_1\}A_1; A_2\{p_3\}.$$

The justification of this rule goes somewhat like this. Say we start evaluating $A_1; A_2$ in state σ where p_1 is true. Now, after evaluation of A_1, we have reached an intermediate state σ' where (due to $\{p_1\}A_1\{p_2\}$) the assertion p_2 holds. Evaluating A_2 in state σ' delivers a final state σ'' where p_3 holds, for $\{p_2\}A_2\{p_3\}$ is true. Thus we have the desired result.

Another rule of inference is the *rule of consequence:*

$$\text{from } p_1 \supset p_2,\, p_3 \supset p_4 \text{ and } \{p_2\}A\{p_3\} \text{ infer } \{p_1\}A\{p_4\},$$

which is obviously valid.

The fact that we allow **goto** statements in our language complicates things. The problem becomes apparent if we take another look at the rule of composition. For instance, if the first statement A_1 in $A \equiv A_1; A_2$ is identical to **goto** L, then the justification of the rule as given above doesn't apply anymore. After evaluation of A_1, the next statement to be executed is not A_2, as was assumed there. Complications are caused by the fact that a statement A can have more than one exit point, namely the normal exit point and the special L-exit points (cf. the discussion after Lemma 4.2).

We can maintain the rule of composition though, if we formulate the meaning of the formula $\{p_1\}A\{p_2\}$ somewhat differently, namely as follows: if A is evaluated beginning in a state where p_1 holds, and evaluation of A terminates *at the normal exit point of* A, in state σ', then p_2 holds in σ'.

Now according to this informal validity definition the formula

$$\{p\}\ \textbf{goto}\ L\{q\} \qquad \ldots(\#)$$

would be valid for every assertion p and q, for evaluation of **goto** L always terminates by "jumping away". However this brings up new problems. For example, the formula

$$\{\textbf{true}\}\ L_1: x:=1;\ \textbf{goto}\ L_2;\ L_2: x:=x\ \{x=0\}$$

would now be derivable, by the following steps

1. $\{\textbf{true}\}\ L_1: x:=1\ \{x=1\}$ (assignment)
2. $\{x=1\}\ \textbf{goto}\ L_2\ \{x=0\}$ (#)
3. $\{x=0\}\ L_2: x:=x\ \{x=0\}$ (assignment)
4. $\{\textbf{true}\}\ L_1: x:=1;\ \textbf{goto}\ L_2;\ L_2: x:=x\ \{x=0\}$ (composition).

But clearly, after evaluation of $L_1: x:=1$; **goto** L_2; $L_2: x:=x$ the postcondition $x=1$ holds.

These difficulties have been solved in [8]. The solution is in essence to put a restriction on the preconditions p allowed in (#), and amounts to the following. Suppose we want to prove $\{p\}S\{q\}$, where $S\equiv L_1: A_1; \ldots; L_n: A_n$. Now assume we can find a list of *label invariants* $p_1, \ldots, p_n$. These p_i are assertions which we assume to be true every time label L_i is reached during execution of S, starting in initial state satisfying p. We now refine our notion of validity once more, and define validity (with respect to the invariants p_i at L_i for $i=1, \ldots, n$) informally as follows:

(*) The formula $\{p\}A\{q\}$ is called valid, iff for every evaluation of A the following holds: if p holds for the initial state, then *either* evaluation terminates at the normal exit point of A and q holds, *or* evaluation terminates at an L_i-exit point of A and p_i holds (for some i, $1\leqq i\leqq n$).

One can see that, according to (*), the formulae $\{p\}$ **goto** $L_i\{q\}$ are no longer valid for all p. Validity holds however for all preconditions p such that $p\supset p_i$. In particular $\{p_i\}$ **goto** $L_i\{\textbf{false}\}$ is valid ($i=1, \ldots, n$). Notice also that the inference rules and the assignment axiom given earlier remain valid according to (*).

Now if we can *derive* $\{p_i\}A_i\{p_{i+1}\}$ using these rules and axioms, and also the formulae $\{p_j\}$ **goto** $L_j\{\textbf{false}\}$, then we know that $\{p_i\}A_i\{p_{i+1}\}$ must be valid according to (*). This means the following: if we consider evaluation of A_i as a sub-statement of $S\equiv L_1: A_1; \ldots; L_n: A_n$, starting at an initial state for which p_i holds, then we can infer from the validity of $\{p_i\}A_i\{p_{i+1}\}$ that at the normal exit point p_{i+1} holds, and that at every L_j-exit point p_j holds. In other words: when evaluation of A_i terminates because label L_j has been reached then the corresponding invariant p_j must hold ($1\leqq j\leqq n$).

But from this we can infer that $\{p_1\}S\{p_{n+1}\}$ holds. For, consider an evaluation of S with initial state satisfying p_1, and suppose that this evaluation terminates. Then this evaluation can be split up in a finite number of subsequent evaluations of sub-statements A_i, and since by the above considerations we are assured that at all "links" labelled L_j the corresponding invariant p_j holds we can infer that p_{n+1} is true when the last evaluation of sub-statement A_n terminates (necessarily at the normal exit point).

The above considerations suggest the following inference rule [8]:

if we can derive $\{p_i\}\, A_i \{p_{i+1}\}$ $(i=1,\dots,n)$ from the assumptions $\{p_j\}$ **goto** L_j $\{\textbf{false}\}$ $(j=1,\dots,n)$, then we may infer $\{p_1\}\, L_1 : A_1; \dots; L_n : A_n \{p_{n+1}\}$.

Now the formula $\{\textbf{true}\}\, S\{x=0\}$, where $S \equiv L_1 : x := 1$; **goto** $L_2; L_2 : x := x$ (sc. the above incorrect derivation) cannot be derived anymore, but a derivation of $\{\textbf{true}\}\, S\{x=1\}$ can be made straightforwardly (take $p_1 \equiv \textbf{true}, p_2 \equiv x=1$).

The inference rule given above leads to compact proofs but, as it stands, is not so suitable for proof-theoretical considerations. Accordingly, we shall now give a more tractable variant of the proof system. In section 8 we shall give a formal justification of the above rule.

It can easily be seen that the assumptions $\{p_j\}$ **goto** L_j $\{\textbf{false}\}$ $(j=1,\dots,n)$ are introduced in the above inference rule only because our proof system must be able to contain information on the label invariants p_i which are used in the proofs. The method that we apply is to take these invariants up in the formulae occurring in the proofs. Our correctness formulae will look like

$$\langle L_1 : p_1, \dots, L_n : p_n \mid \{p\}\, A\{q\}\rangle,$$

so the invariants p_i corresponding to L_i are supplied explicitly in our formulae, instead of implicitly in the assumptions used in a proof. The informal meaning of the above formula is the one as given by $(*)$.

After this introduction the following definitions must be clear.

Definition (Syntax of correctness formulae). The class $\mathscr{Invl}$ *(list of label invariants)* with typical element D is defined by

$$D ::= L : p \mid L : p, D$$

where it is required that if $D \equiv L_1 : p_1, \dots, L_n : p_n$, then $L_i \not\equiv L_j$ for $i \neq j$. We write $[L_i : p_i]_{i=1}^n$ instead of $L_1 : p_1, \dots, L_n : p_n$.

We say $(L : p)$ *occurs in* D (notation: $(L : p)$ **in** D) iff $L \equiv L_j$, $p \equiv p_j$ and $D \equiv [L_i : p_i]_{i=1}^n$ for some j $(1 \leqq j \leqq n)$.

The class $\mathscr{Corr}$ *(correctness formulae)* with typical element f is defined by

$$f ::= p \mid \langle D; \{p\}\, A\{q\}\rangle \mid \{p\}\, S\{q\}.$$

We write $\langle D \mid \{p\}\, A\{q\}\rangle$ instead of $\langle D; \{p\}\, A\{q\}\rangle$.

Definition (proof system $\mathscr{H}$). The axioms of $\mathscr{H}$ are given by the following schemes:

(A1) $\langle D \mid \{p[s/x]\}\, x := s\{p\}\rangle$

(A2) $\langle D \mid \{p\}$ **goto** $L\{\textbf{false}\}\rangle$,
where $D \equiv [L_i : p_i]_{i=1}^n$, $L \equiv L_j$, $p \equiv p_j$ for some j $(1 \leqq j \leqq n)$.

(A3) p,
where p is a valid assertion (i.e. $\forall \sigma \in \Sigma_0 : \mathscr{T}[\![p]\!]\sigma = tt$).

The rules of inference have the form

$$\frac{f_1, \ldots, f_n}{f_{n+1}}$$

("from $f_1, \ldots,$ and f_n, infer f_{n+1}") and are given by the following schemes:

(R1) $$\frac{p_1 \supset p_2, p_3 \supset p_4, \langle D | \{p_2\} A \{p_3\} \rangle}{\langle D | (p_1\} A \{p_4\} \rangle}$$

(R2) $$\frac{p_1 \supset p_2, p_3 \supset p_4, \{p_2\} S \{p_3\}}{\{p_1\} S \{p_4\}}$$

(R3) $$\frac{\langle D | \{p_1\} A \{p_2\} \rangle, \langle D | \{p_2\} A' \{p_3\} \rangle}{\langle D | \{p_1\} A; A' \{p_3\} \rangle}$$

(R4) $$\frac{\langle \mathrm{D} | \{p \wedge b\} A \{q\} \rangle \langle D | \{p \wedge \neg b\} A' \{q\} \rangle}{\langle D | \{p\} \textbf{ if } b \textbf{ then } A \textbf{ else } A' \textbf{ fi } \{q\} \rangle}$$

(R5) $$\frac{\langle D | \{p_1\} A_1 \{p_2\} \rangle, \ldots, \langle D | \{p_n\} A_n \{p_{n+1}\}}{\{p_1\} [L_i : A_i]_{i=1}^{n} \{p_{n+1}\}} \quad \text{where } D \equiv [L_i : p_i]_{i=1}^{n}.$$

Definition (normal pair, normal correctness formula, normal fragment of $\mathscr{H}$). A pair $\langle D, A \rangle$ is called *normal* if all labels in A occur in D.

A correctness formula f is called *normal* if *either* f is an assertion, *or* $f \equiv \langle D | \{p\} A \{q\} \rangle$ and $\langle D, A \rangle$ is a normal pair, or $f \equiv \{p\} S \{q\}$ and S is a normal program (i.e. all labels in S are declared).

The *normal fragment* of the proof system $\mathscr{H}$, denoted by $\mathscr{H}_N$, is the system $\mathscr{H}$ restricted to normal formulae only.

Definition (formal proof). Let $f \in \mathscr{Corr}$. A sequence $f_1, \ldots, f_n$ with $f_i \in \mathscr{Corr}$ ($i = 1, \ldots, n$) is called *a formal proof of* f in $\mathscr{H}$ if

a) $f \equiv f_n$
b) for all f_i with $1 \leqq i \leqq n$ the following holds:
   either 1) f_i is (an instance of) an axiom
   or 2) there exist $f_{i_1}, \ldots, f_{i_k} \in \mathscr{Corr}$ with $1 \leqq i_j < i$ for $1 \leqq j \leqq k$, such that

$$\frac{f_{i_1}, \ldots, f_{i_k}}{f_i}$$

is (an instance of) a rule of inference.

We say that f is *provable*, notation $\vdash f$, if there exists a formal proof of f.

The system defined above is dependent on the interpretation $\mathscr{I}$ of the primitive relation and function symbols, because the axioms of (A3) are determined by $\mathscr{T}$, which function depends on $\mathscr{I}$. We include all true assertions as axioms because we don't want to pay attention to deduction systems for the assertions only. We want to focus on the rules which can be used to prove properties of statements and programs. Also, in the proof of completeness of our system ("every valid formula is provable") we don't want to be hindered by deduction systems for the assertions which are possibly incomplete.

We now turn to the question of validity of correctness formulae (again with respect to an interpretation $\mathscr{I}$). We use the notation $\models f$ to denote that f is valid. An informal definition of the concept has been given in the remarks preceding the definition of the deduction system. We will now formalize the ideas developed there. By now it must be clear that in the validity definition the semantic function $\mathscr{A}$ will be much easier in use than the function $\mathscr{N}$ (see the remarks preceding the definition of $\mathscr{A}$ at the end of Chap. 4).

Definition (Validity). *Validity* of a correctness formula f, notation $\models f$, is defined by

a) $\models p$ iff $\forall \sigma \in \Sigma_0\colon \mathscr{T}[\![p]\!]\sigma = tt$

b) $\models \langle D | \{p\} A \{q\}\rangle$ iff

$$\forall \sigma \in \Sigma_0\colon \mathscr{T}[\![p]\!]\sigma = tt$$
$$\Rightarrow \begin{bmatrix} (\exists \sigma' \in \Sigma_0\colon \mathscr{A}[\![A]\!]\sigma = \sigma' \wedge \mathscr{T}[\![q]\!]\sigma' = tt) \vee \\ (\exists \sigma' \in \Sigma_0 \, \exists (L\colon p') \textbf{ in } D\colon \mathscr{A}[\![A]\!]\sigma = \langle \sigma', L\rangle \wedge \mathscr{T}[\![p']\!]\sigma' = tt) \end{bmatrix}$$

c) $\models \{p\} S \{q\}$ iff $\forall \gamma \in \Gamma \ \forall \sigma, \sigma' \in \Sigma_0\colon [(\mathscr{T}[\![p]\!]\sigma = tt \wedge \sigma' = \mathscr{M}[\![S]\!]\gamma\sigma) \Rightarrow \mathscr{T}[\![q]\!]\sigma' = tt]$.

In words this amounts to the following. An assertion p is valid if it is true in all (defined) states. A formula $\{p\} S \{q\}$ is valid, if evaluation of S with initial state σ satisfying p, either doesn't terminate or terminates in final state σ' for which q holds. The most complicated case is $f \equiv \langle D | \{p\} A \{q\}\rangle$. This f is valid if for every state σ satisfying p the following holds: if evaluation of A terminates normally in σ' then we want q to be true in σ'; if evaluation terminates by a jump to some L in state σ', we want this L to be an L_j in $D \equiv [L_i\colon p_i]_{i=1}^n$, and the corresponding assertion p_j must be true in σ'.

7. Soundness and Completeness of $\mathscr{H}_N$

We next like to show that the deduction system is sound ("$\vdash f \Rightarrow \models f$"), and complete ("$\models f \Rightarrow \vdash f$"). Now the definition of provability as well as that of validity shows that both notions are dependent on the interpretation $\mathscr{I}$ chosen. In this section we will prove that $\vdash f \Rightarrow \models f$ holds for all correctness formulae. The converse is not true in general. Following Cook [9] we have to put a restriction on the interpretations allowed: only those interpretations are taken into account which make the class $\mathscr{A}\!ssn$ *expressive with respect to the language* $\mathscr{P}\!rog$. Only if $\mathscr{A}\!ssn$ is expressive we can be assured that it is possible to find suitable label invariants $p_1, \ldots, p_n \in \mathscr{A}\!ssn$ for every program $S \equiv [L_i\colon A_i]_{i=1}^n$. The completeness theorem to be proved will then be that under every interpretation $\mathscr{I}$ such that $\mathscr{A}\!ssn$ is expressive with respect to $\mathscr{P}\!rog$ we have that $\models f \Rightarrow \vdash f$ for every normal correctness formula f.

Definition (validity of rules of inference). A rule of inference

$$\frac{f_1, \ldots, f_n}{f_{n+1}}$$

is called *valid* if $(\models f_1, \ldots, \models f_n) \Rightarrow \models f_{n+1}$.

Note that validity of an inference rule again depends on the underlying interpretation $\mathscr{I}$ just like the validity of a correctness formula.

Lemma 7.1. *Every axiom and every rule of inference in $\mathscr{H}$ is valid.*

Proof. (A1) We have to prove $\models \langle D | \{p[s/x]\}\, x := s\, \{p\}\rangle$. We have $\mathscr{A}[\![x := s]\!]\,\sigma = \sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\}$ for all $\sigma \in \Sigma_0$. Furthermore $\mathscr{T}[\![p[s/x]]\!]\,\sigma = tt$ implies $\mathscr{T}[\![p]\!](\sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\}) = tt$ by Lemma 6.1d. Thus we have that for all $\sigma \in \Sigma_0$ with $\mathscr{T}[\![p[s/x]]\!]\,\sigma = tt$ there is a σ', namely $\sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\}$, such that $\mathscr{A}[\![x := s]\!]\,\sigma = \sigma'$ and $\mathscr{T}[\![p]\!]\,\sigma' = tt$.

(A2) We have to prove $\models \langle D | \{p_j\}$ **goto** L_j $\{$**false**$\}\rangle$ for $D \equiv [L_i : p_i]_{i=1}^n$. Choose $\sigma \in \Sigma_0$ such that $\mathscr{T}[\![p_j]\!]\,\sigma = tt$. We have $\mathscr{A}[\![$**goto** $L_j]\!]\,\sigma = \langle \sigma, L_j\rangle$. Thus there is a σ', namely σ itself and a pair $(L : p'')$ in D, namely $(L_j : p_j)$, such that $\mathscr{A}[\![$**goto** $L_j]\!]\,\sigma = \langle \sigma', L\rangle$ and $\mathscr{T}[\![p'']\!]\,\sigma' = tt$.

(A3) Evident.

(R1) Suppose $\models p_1 \supset p_2$, $\models p_3 \supset p_4$ and $\models \langle D | \{p_2\}\, A\, \{p_3\}\rangle$. We want to prove $\models \langle D | \{p_1\}\, A\, \{p_4\}\rangle$. Choose a $\sigma \in \Sigma_0$, and assume $\mathscr{T}[\![p_1]\!]\,\sigma = tt$. From $p_1 \supset p_2$ we infer $\mathscr{T}[\![p_2]\!]\,\sigma = tt$. The fact that $\models \langle D | \{p_2\}\, A\, \{p_3\}$ holds yields

either $\exists \sigma' \in \Sigma_0$: $\mathscr{A}[\![A]\!]\,\sigma = \sigma' \wedge \mathscr{T}[\![p_3]\!]\,\sigma' = tt$. But in this case we can use $\models p_3 \supset p_4$ to infer $\exists \sigma' \in \Sigma_0$: $\mathscr{A}[\![A]\!]\,\sigma = \sigma' \wedge \mathscr{T}[\![p_4]\!]\,\sigma' = tt$ ...(*)

or $\exists \sigma' \in \Sigma_0$: $\exists (L : p'')$ **in** D: $\mathscr{A}[\![A]\!]\,\sigma = \langle \sigma', L\rangle \wedge \mathscr{T}[\![p'']\!]\,\sigma' = tt$ ...(**)

But now we have proved $\mathscr{T}[\![p_1]\!]\,\sigma = tt \Rightarrow (*) \vee (**)$, and we conclude that $\models \langle D | \{p_1\}\, A\, \{p_4\}\rangle$ holds.

(R2) Analogously.

(R3) Suppose $\models \langle D | \{p_1\}\, A\, \{p_2\}\rangle$ and $\models \langle D | \{p_2\}\, A'\, \{p_3\}\rangle$. We have to prove $\models \langle D | \{p_1\}\, A; A'\, \{p_3\}\rangle$. Choose a $\sigma \in \Sigma_0$ such that $\mathscr{T}[\![p_1]\!]\,\sigma = tt$. From $\models \langle D | \{p_1\}\, A\, \{p_2\}\rangle$ we infer that

either $(\mathscr{A}[\![A]\!]\,\sigma = \sigma' \wedge \mathscr{T}[\![p_2]\!]\,\sigma' = tt)$ for some $\sigma' \in \Sigma_0$ ...(1)

or $(\mathscr{A}[\![A]\!]\,\sigma = \langle \sigma', L\rangle \wedge \mathscr{T}[\![p'']\!]\,\sigma' = tt)$ for some $\sigma' \in \Sigma_0$, $(L : p'')$ **in** D ...(2)

ad (1). $\models \langle D | \{p_2\}\, A'\, \{p_3\}\rangle$ and $\mathscr{T}[\![p_2]\!]\,\sigma' = tt$ for some $\sigma' \in \Sigma_0$ give us:

either $(\mathscr{A}[\![A']\!]\,\sigma' = \sigma'' \wedge \mathscr{T}[\![p_3]\!]\,\sigma'' = tt)$ for some $\sigma'' \in \Sigma_0$. From $\mathscr{A}[\![A]\!]\,\sigma = \sigma'$ and $\mathscr{A}[\![A']\!]\,\sigma' = \sigma''$ we infer $\mathscr{A}[\![A; A']\!]\,\sigma = \sigma''$. Furthermore we have $\mathscr{T}[\![p_3]\!]\,\sigma'' = tt$,

or $(\mathscr{A}[\![A']\!]\,\sigma' = \langle \sigma'', L\rangle \wedge \mathscr{T}[\![p'']\!]\,\sigma'' = tt)$ for some $\sigma'' \in \Sigma_0$ and some pair $(L : p'')$ **in** D. But then $\mathscr{A}[\![A]\!]\,\sigma = \sigma'$ and $\mathscr{A}[\![A']\!]\,\sigma' = \langle \sigma'', L\rangle$ give us $\mathscr{A}[\![A; A']\!]\,\sigma = \langle \sigma'', L\rangle$ and we have also $\mathscr{T}[\![p'']\!]\,\sigma'' = tt$.

ad (2). From $\mathscr{A}[\![A]\!]\,\sigma = \langle \sigma', L\rangle$ we have $\mathscr{A}[\![A; A']\!]\,\sigma = \langle \sigma', L\rangle$. Furthermore we have have $\mathscr{T}[\![p'']\!]\,\sigma' = tt$.

The conclusion is that for every choice of σ the conditions imposed by the definition of $\models \langle D|\{p_1\}A; A'\{p_3\}\rangle$ are satisfied.

(R 4) can be proved analogously using results like

$$(\mathscr{A}[\![A]\!]\,\sigma=\sigma' \wedge \mathscr{W}[\![b]\!]\,\sigma=tt) \Rightarrow \mathscr{A}[\![\textbf{if } b \textbf{ then } A \textbf{ else } A' \textbf{ fi}]\!]\,\sigma=\sigma'.$$

(R 5) Suppose $\models \langle D|\{p_i\}A_i\{p_{i+1}\}\rangle$ $(i=1,\dots,n)$, where $D\equiv[L_i:p_i]_{i=1}^n$. We have to prove $\models \{p_1\}[L_i:A_i]_{i=1}^n\{p_{n+1}\}$, or equivalently

$$\forall\gamma\in\Gamma\ \ \forall\sigma,\sigma'\in\Sigma_0[(\mathscr{T}[\![p_1]\!]\,\sigma=tt\wedge\mathscr{M}[\![[L_i:A_i]_{i=1}^n]\!]\,\gamma\sigma=\sigma')\Rightarrow\mathscr{T}[\![p_{n+1}]\!]\,\sigma'=tt].$$

So, choose $\gamma\in\Gamma$, and let ϕ_i, $\phi_i^{(k)}$ and $\gamma^{(k)}$ be derived from $[L_i:A_i]_{i=1}^n$ and γ as in Lemma 4.2. We now prove the following lemma.

Lemma. $\forall k\in\mathbb{N}$ $[\forall\sigma,\sigma'\in\Sigma$: $(\mathscr{T}[\![p_i]\!]\,\sigma=tt\wedge\sigma'=\phi_i^{(k)}\sigma)\Rightarrow\mathscr{T}[\![p_{n+1}]\!]\,\sigma'=tt$, *for* $i=1,\dots,n+1]$.

Proof (induction on k). *Basis* $(k=0)$. This is easy, because (i) $\phi_i^{(0)}=\lambda\sigma\cdot\bot$ for $i=1,\dots,n$ and therefore there is no $\sigma'\in\Sigma_0$ such that $\sigma'=\phi_i^{(0)}\sigma$; (ii) $\phi_{n+1}^{(i)}=\lambda\sigma\cdot\sigma$, but then the assumption reduces to $\mathscr{T}[\![p_{n+1}]\!]\,\sigma=tt\wedge\sigma'=(\lambda\sigma\cdot\sigma)\sigma=\sigma$, and thus the conclusion $\mathscr{T}[\![p_{n+1}]\!]\,\sigma'=tt$ holds.

Induction Step. Choose an i $(1\leqq i\leqq n$; the case $i=n+1$ is again trivial) and choose $\sigma,\sigma'\in\Sigma_0$ such that $\mathscr{T}[\![p_i]\!]\,\sigma=tt$ and $\sigma'=\phi_i^{(k+1)}\sigma$. Now $\phi_i^{(k+1)}\sigma=\mathscr{N}[\![A_i]\!]\,\gamma^{(k)}\phi_i^{(k)}\sigma$. From $\models\langle D|\{p_i\}A_i\{p_{i+1}\}\rangle$ we know that

either $\mathscr{A}[\![A_i]\!]\,\sigma=\sigma''$ and $\mathscr{T}[\![p_{i+1}]\!]\,\sigma''=tt$, for some $\sigma''\in\Sigma_0$. But then we have $\sigma'=\phi_i^{(k+1)}\sigma=\phi_{i+1}^{(k)}\sigma''$ (using 4.3.1°). Induction hypothesis, and $\mathscr{T}[\![\mathrm{p}_{i+1}]\!]\,\sigma''=\mathrm{tt}$ yield $\mathscr{T}[\![p_{n+1}]\!]\,\sigma'=tt$,

or $\mathscr{A}[\![A_i]\!]\,\sigma=\langle\sigma'',L_j\rangle$ and $\mathscr{T}[\![p_j]\!]\,\sigma''=tt$, for some $\sigma''\in\Sigma_0$ and some j $(1\leqq j\leqq n)$. Now $\sigma'=\phi_i^{(k+1)}\sigma=\mathscr{N}[\![A_i]\!]\,\gamma^{(k)}\phi_{i+1}^{(k)}\sigma=\gamma^{(k)}[\![L_j]\!]\,\sigma''=\phi_j^{(k)}\sigma''$ (using (4.3.2°). Induction hypothesis and $\mathscr{T}[\![p_j]\!]\,\sigma''=tt$ yield $\mathscr{T}[\![p_{n+1}]\!]\,\sigma'=tt$.

This proves the lemma. □

Now, returning to the proof of $\models\{p_1\}[L_i: A_i]_{i=1}^n\{p_{n+1}\}$, we have by definition that $\mathscr{M}[\![[L_i: A_i]_{i=1}^n]\!]\,\gamma=\mathscr{N}[\![A_1;\dots;A_n]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\{\lambda\sigma\cdot\sigma\}=\phi_1$ by Lemma 4.2.2. And $\phi_1=\bigsqcup_k\phi_1^{(k)}$, by Lemma 4.2.1.

Choose $\sigma,\sigma'\in\Sigma_0$ such that $\mathscr{T}[\![p_1]\!]\,\sigma=tt$ and $\sigma'=\phi_1\sigma$. Then (because $\phi_1\sigma=\bigsqcup_k(\phi_1^{(k)}\sigma)$) there is a $\bar{k}$ such that $\sigma'=\phi_1^{(\bar{k})}\sigma$, and the lemma gives us $\mathscr{T}[\![p_{n+1}]\!]\,\sigma'=tt$.

This proves $\models\{p_1\}[L_i:A_i]_{i=1}^n\{p_{n+1}\}$, which was the last clause in the proof of 7.1. □

Theorem 7.2. *The proof system $\mathscr{H}$ is sound, i.e. for every interpretation $\mathscr{I}$ and every correctness formula f we have $\vdash f\Rightarrow\models f$.*

Proof. Induction on the length of the proof of f, using Lemma 7.1. □

We now turn our attention to the question of completeness of the proof system, i.e., $\models f\Rightarrow\vdash f$. If f is an assertion p, then we can simply use axiom scheme (A 3), so there is no problem here.

The next possibility is $f \equiv \langle D | \{p\} A \{q\} \rangle$. Suppose this f is valid. Now we have to construct a formal proof of this formula. This will be done using the concept of *weakest precondition:* we will show that for all $D \in \mathscr{I}\mathit{nvl}$, $A \in \mathscr{S}\mathit{tat}$ and $p \in \mathscr{A}\mathit{ssn}$ (such that $\langle D, A \rangle$ is normal), we can construct a $q \in \mathscr{A}\mathit{ssn}$ which is the weakest formula that makes $\langle D | \{q\} A \{p\} \rangle$ valid (by "weakest" we mean true in as many states as possible). This is part of Lemma 7.4.

In the same lemma we show that for this assertion q the formula $\langle D | \{q\} A \{p\} \rangle$ is provable. Once we have reached this result, the rest is easy. We use the property that, if q expresses the weakest precondition of A with respect to p and D, and if $\models \langle D | \{p'\} A \{p\} \rangle$ for some $p' \in \mathscr{A}\mathit{ssn}$, then we have $\models p' \supset q$ (otherwise the precondition q wouldn't be the weakest one). Thus in this case we can derive $\langle D | \{p'\} A \{p\} \rangle$ using (R1).

Definition (weakest precondition). Let $A \in \mathscr{S}\mathit{tat}$, $p \in \mathscr{A}\mathit{ssn}$, $D \in \mathscr{I}\mathit{nvl}$. We say that *q expresses the weakest precondition of A with respect to postcondition p and invariant list D* iff

$$\forall \sigma \in \Sigma_0 \colon \mathscr{T}[\![q]\!]\sigma = tt$$
$$\Leftrightarrow \begin{bmatrix} (\exists \sigma' \in \Sigma_0 \colon [\mathscr{A}[\![A]\!]\sigma = \sigma' \wedge \mathscr{T}[\![p]\!]\sigma' = tt]) \vee \\ (\exists \sigma' \in \Sigma_0 \ \exists (L \colon p') \ \mathbf{in}\ D \colon [\mathscr{A}[\![A]\!]\sigma = \langle \sigma', L \rangle \wedge \mathscr{T}[\![p']\!]\sigma' = tt]) \end{bmatrix}.$$

We write $p \simeq wp[\![A, p, D]\!]$ to express this.

Lemma 7.3. *Let $A \in \mathscr{S}\mathit{tat}$, $p, q \in \mathscr{A}\mathit{ssn}$, $D \in \mathscr{I}\mathit{nvl}$. If $q \simeq wp[\![A, p, D]\!]$, then*

a) $\models \langle D | \{q\} A \{p\} \rangle$ *(i.e., q is a precondition)*

b) $\forall p' \in \mathscr{A}\mathit{ssn} \colon [\models \langle D | \{p'\} A \{p\} \rangle \Rightarrow \ \models p' \supset q]$ *(q is the weakest).*

Proof. Immediate from the definitions. □

Lemma 7.4. *For all $A \in \mathscr{S}\mathit{tat}$, $p \in \mathscr{A}\mathit{ssn}$, $D \in \mathscr{I}\mathit{nvl}$ such that $\langle D, A \rangle$ is normal, we can find $q \in \mathscr{A}\mathit{ssn}$ for which $q \simeq wp[\![A, p, D]\!]$. Moreover for this q we also have $\vdash \langle D | \{q\} A \{p\} \rangle$ in $\mathscr{H}_N$.*

Proof. By induction on the structure of A. We distinguish four cases.

1°. $A \equiv x := s$. Choose $p \in \mathscr{A}\mathit{ssn}$ and $D \in \mathscr{I}\mathit{nvl}$. Then $p[s/x] \simeq wp[\![x := s, p, D]\!]$. For, choose $\sigma \in \Sigma_0$. We have to show

$$\mathscr{T}[\![p[s/x]]\!]\sigma = tt \Leftrightarrow (\exists \sigma' \in \Sigma_0 [(\mathscr{A}[\![x := s]\!]\sigma = \sigma') \wedge \mathscr{T}[\![p]\!]\sigma' = tt]) \vee$$
$$(\exists \sigma' \in \Sigma_0 \ \exists (L \colon p'') \ \mathbf{in}\ D[(\mathscr{A}[\![A]\!]\sigma = \langle \sigma', L \rangle) \wedge \mathscr{T}[\![p'']\!]\sigma' = tt]).$$

Now $\mathscr{A}[\![x := s]\!]\sigma = \sigma\{\mathscr{V}[\![s]\!]\sigma/x\} \in \Sigma_0$, so the above equivalence comes down to

$$\mathscr{T}[\![p[s/x]]\!]\sigma = tt \Leftrightarrow \exists \sigma' \in \Sigma_0 [(\mathscr{A}[\![x := s]\!]\sigma = \sigma') \wedge \mathscr{T}[\![p]\!]\sigma' = tt]$$
$$\Leftrightarrow \mathscr{T}[\![p]\!](\sigma\{\mathscr{V}[\![s]\!]\sigma/x\}) = tt,$$

and this is 6.2d.

Furthermore, we have $\vdash \langle D | \{p[s/x]\} x := s \{p\} \rangle$ by (A1).

2°. $A \equiv A_1; A_2$. Choose $p \in \mathscr{A}\mathit{ssn}$ and $D \in \mathscr{I}\mathit{nvl}$ such that $\langle D, A \rangle$ is normal. By induction there is a $q' \in \mathscr{A}\mathit{ssn}$ with $q' \simeq wp[\![A_2, p, D]\!]$ and $\vdash \langle D | \{q'\} A_2 \{p\} \rangle$

in $\mathscr{H}_N$. Again by induction we have $q \in \mathscr{A}ssn$ such that $q \simeq wp[\![A_1, q'; D]\!]$ and $\vdash \langle D | \{q\} A_1 \{q'\}\rangle$ in $\mathscr{H}_N$.

First, we will show that for this q we have $q \simeq wp[\![A_1; A_2, p, D]\!]$. Choose a $\sigma \in \Sigma_0$. We have to prove

$$\begin{aligned}\mathscr{T}[\![q]\!]\,\sigma = tt \Leftrightarrow {} & (\exists\, \sigma' \in \Sigma_0 [\mathscr{A}[\![A_1; A_2]\!]\,\sigma = \sigma' \wedge \mathscr{T}[\![p]\!]\,\sigma' = tt]) \vee \\ & (\exists\, \sigma' \in \Sigma_0\ \exists (L : p')\ \textbf{in}\ D[\mathscr{A}[\![A_1; A_2]\!]\,\sigma = \langle \sigma', L\rangle \wedge \mathscr{T}[\![p']\!]\,\sigma' = tt]).\end{aligned}$$

We distinguish two cases:

a) $\mathscr{A}[\![A_1]\!]\,\sigma = \sigma''$. We then have $\mathscr{A}[\![A_1; A_2]\!]\,\sigma = \mathscr{A}[\![A_2]\!]\,\sigma''$ by definition of $\mathscr{A}$. Using these facts the above equivalence reduces to

$$\begin{aligned}\mathscr{T}[\![q]\!]\,\sigma = tt \Leftrightarrow {} & (\exists\, \sigma' \in \Sigma_0 [\mathscr{A}[\![A_2]\!]\,\sigma'' = \sigma' \wedge \mathscr{T}[\![p]\!]\,\sigma' = tt]) \vee \\ & (\exists\, \sigma' \in \Sigma_0\ \exists (L : p')\ \textbf{in}\ D[\mathscr{A}[\![A_2]\!]\,\sigma'' = \langle \sigma', L\rangle \wedge \mathscr{T}[\![p']\!]\,\sigma' = tt]),\end{aligned}$$

and by $q' \simeq wp[\![A_2, p, D]\!]$ this is equivalent to $\mathscr{T}[\![q]\!]\,\sigma = tt \Leftrightarrow \mathscr{T}[\![q']\!]\,\sigma'' = tt$. Now we have by $q \simeq wp[\![A_1, q', D]\!]$

$$\begin{aligned}\mathscr{T}[\![q]\!]\,\sigma = tt \Leftrightarrow {} & (\exists\, \sigma' \in \Sigma_0 [\mathscr{A}[\![A_1]\!]\,\sigma = \sigma' \wedge \mathscr{T}[\![q']\!]\,\sigma' = tt]) \vee \\ & (\exists\, \sigma' \in \Sigma_0\ \exists (L : p')\ \textbf{in}\ D[\mathscr{A}[\![A_1]\!]\,\sigma = \langle \sigma', L\rangle \wedge \mathscr{T}[\![p']\!]\,\sigma' = tt]).\end{aligned}$$

Substituting σ'' for $\mathscr{A}[\![A_1]\!]\,\sigma$ (that is the assumption) the right-hand side of the equivalence reduces to $\mathscr{T}[\![q']\!]\,\sigma'' = tt$, and we are ready.

b) $\mathscr{A}[\![A_1]\!]\,\sigma = \langle \sigma'', L'\rangle$. We then also have that $\mathscr{A}[\![A_1; A_2]\!]\,\sigma = \langle \sigma'', L'\rangle$ and the definition of $q \simeq wp[\![A_1; A_2, p, D]\!]$ reduces to

$$\mathscr{T}[\![q]\!]\,\sigma = tt \Leftrightarrow \exists p'' \in \mathscr{A}ssn\,[(L' : p'')\ \textbf{in}\ D \wedge \mathscr{T}[\![p'']\!]\,\sigma'' = tt].$$

But this equivalence is immediate from $q \simeq wp[\![A_1, q', D]\!]$ and $\mathscr{A}[\![A_1]\!]\,\sigma = \langle \sigma'', L'\rangle$. So, we have proved that $q \simeq wp[\![A_1; A_2, p, D]\!]$.

The proof that $\langle D | \{q\} A_1; A_2 \{p\}\rangle$ can be derived in $\mathscr{H}_N$ is easy by the assumptions on q and q' (using rule (R 3) of composition), and the fact that the pair $\langle D, A_1; A_2\rangle$ is normal (which means that $\langle D | \{p_1\} A_1 \{p_2\}\rangle$ and $\langle D | \{p_2\} A_2 \{p_3\}\rangle$ are normal formulae).

3°. $A \equiv$ **if** b **then** A_1 **else** A_2 **fi**. Choose $p \in \mathscr{A}ssn$ and $D \in \mathscr{I}mvl$, such that $\langle D, A\rangle$ is normal. By induction we ave $q_1, q_2 \in \mathscr{A}ssn$ such that

$$\begin{aligned} & q_1 \simeq wp[\![A_1, p, D]\!] \quad \text{and} \quad \vdash \langle D | \{q_1\} A_1 \{p\}\rangle \quad \text{in } \mathscr{H}_N \\ & q_2 \simeq wp[\![A_2, p, D]\!] \quad \text{and} \quad \vdash \langle D | \{q_2\} A_2 \{p\}\rangle \quad \text{in } \mathscr{H}_N.\end{aligned}$$

We will show that for $q \equiv$ **if** b **then** q_1 **else** q_2 **fi** we have

a) $q \simeq wp[\![\textbf{if}\ b\ \textbf{then}\ A_1\ \textbf{else}\ A_2\ \textbf{fi},\ p, D]\!]$

b) $\langle D | \{q\}\ \textbf{if}\ b\ \textbf{then}\ A_1\ \textbf{else}\ A_2\ \textbf{fi}\ \{p\}\rangle$ in $\mathscr{H}_N$.

a) Choose a $\sigma \in \Sigma_0$. Without loss of generality $\mathscr{W}[\![b]\!]\,\sigma = tt$. Then

$$\mathscr{T}[\![\textbf{if}\ b\ \textbf{then}\ q_1\ \textbf{else}\ q_2\ \textbf{fi}]\!]\,\sigma = tt \Leftrightarrow \mathscr{T}[\![q_1]\!]\,\sigma = tt.$$

Also $\mathscr{A}[\![\textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}]\!]\sigma = \mathscr{A}[\![A_1]\!]\sigma$. Now $q_1 \simeq wp[\![A_1, p, D]\!]$ is equivalent to

$$\begin{aligned}\mathscr{T}[\![q_1]\!]\sigma = tt \Leftrightarrow {} & (\exists \sigma' \in \Sigma_0 [\mathscr{A}[\![A_1]\!]\sigma = \sigma' \wedge \mathscr{T}[\![p]\!]\sigma' = tt]) \vee \\ & (\exists \sigma' \in \Sigma_0\ \exists (L : p') \textbf{ in } D[\mathscr{A}[\![A_1]\!]\sigma = \langle \sigma', L\rangle \wedge \mathscr{T}[\![p']\!]\sigma' = tt]).\end{aligned}$$

Combining these results, we get

$$\begin{aligned}&\mathscr{T}[\![\textbf{if } b \textbf{ then } q_1 \textbf{ else } q_2 \textbf{ fi}]\!]\sigma = tt \Leftrightarrow \\ &(\exists \sigma' \in \Sigma_0 [\mathscr{A}[\![\textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}]\!]\sigma = \sigma' \wedge \mathscr{T}[\![p]\!]\sigma' = tt]) \vee \\ &(\exists p' \in \Sigma_0 \exists (L : p') \textbf{ in } D[\mathscr{A}[\![\textbf{if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi}]\!]\sigma = \langle \sigma', L\rangle \wedge \mathscr{T}[\![p']\!]\sigma' = tt])\end{aligned}$$

and this is the result we were aiming at.

b) We have $q \wedge b \equiv (\textbf{if } b \textbf{ then } q_1 \textbf{ else } q_2 \textbf{ fi}) \wedge b$, and thus $\models q \wedge b \supset q_1$. Also, using (A3) and (R1), and $\vdash \langle D | \{q_1\} A \{p\}\rangle$ (in $\mathscr{H}_N$) we get $\vdash \langle D | \{q \wedge b\} A_1 \{p\}\rangle$ in $\mathscr{H}_N$. Analogously $\vdash \langle D | \{q \wedge \neg b\} A_2 \{p\}\rangle$ in $\mathscr{H}_N$. So, by inference rule (R4): $\vdash \langle D | \{q\} \textbf{ if } b \textbf{ then } A_1 \textbf{ else } A_2 \textbf{ fi } \{p\}\rangle$ in $\mathscr{H}_N$.

4°. $A \equiv \textbf{goto } L$. Choose $p \in \mathscr{A}ssn$ and $D \in \mathscr{I}nvl$, such that $\langle D, A\rangle$ is normal. This means that we have a $q \in \mathscr{A}ssn$ such that $(L : q)$ **in** D. We prove $q \simeq wp[\![\textbf{goto } L, p, D]\!]$. We have to prove

$$\begin{aligned}\mathscr{T}[\![q]\!]\sigma = tt \Leftrightarrow {} & (\exists \sigma' \in \Sigma_0 [\mathscr{A}[\![\textbf{goto } L]\!]\sigma = \sigma' \wedge \mathscr{T}[\![p]\!]\sigma' = tt] \vee \\ & (\exists \sigma' \in \Sigma_0\ \exists (L' : p') \textbf{ in } D[\mathscr{A}[\![\textbf{goto } L]\!]\sigma = \langle \sigma', L'\rangle \wedge \mathscr{T}[\![p']\!]\sigma' = tt]).\end{aligned}$$

Because $\mathscr{A}[\![\textbf{goto } L]\!]\sigma = \langle \sigma, L\rangle \in \Sigma_0 \times \mathscr{L}var$, the equivalence reduces to

$$\mathscr{T}[\![q]\!]\sigma = tt \Leftrightarrow \exists p' \in \mathscr{A}ssn[(L : p') \textbf{ in } D \wedge \mathscr{T}[\![p']\!]\sigma = tt].$$

Now we have $(L : q)$ **in** D and thus the right-hand side is equivalent to $\mathscr{T}[\![q]\!]\sigma = tt$.

Furthermore, in order to show that $\vdash \langle D | \{q\} \textbf{ goto } L\{p\}\rangle$ in $\mathscr{H}_N$, we have by (A2) $\vdash \langle D | \{q\} \textbf{ goto } L\{\textbf{false}\}\rangle$ and by (A3) $\vdash \textbf{false} \supset p$. So we can use (R1) to derive $\vdash \langle D | \{q\} \textbf{ goto } L\{p\}\rangle$ in $\mathscr{H}_N$.

This completes the proof of Lemma 7.4. □

Observe that in this lemma it is not merely proved that there exists a formula q expressing the weakest precondition for any A, p and D, such that $\langle D, A\rangle$ is normal. The proof also provides a purely syntactical method to derive such a formula. This shows that for every A, p and D with the above restriction, we can *construct* an assertion q expressing the weakest precondition. Thus this q is independent of the interpretation $\mathscr{I}$ of the primitive relation and function symbols.

Note also that there are many assertions expressing the weakest precondition of A with respect to p and D. For instance, if $q \simeq wp[\![A, p, D]\!]$ then the same holds for $q \wedge \textbf{true}$ (to give a trivial example).

We now state and prove the completeness result for correctness formulae having the form $\langle D | \{p\} A\{q\}\rangle$.

Lemma 7.5. *For all $A \in \mathcal{S}tat$, $p, q \in \mathcal{A}ssn$ and $D \in \mathcal{I}mvl$ such that $\langle D, A\rangle$ is normal we have*

$$\models \langle D \mid \{p\}\, A\{q\}\rangle \Rightarrow\ \vdash \langle D \mid \{p\}\, A\{q\}\rangle \quad \text{in } \mathcal{H}_N.$$

Proof. Choose A, p, q and D such that the assumptions are true. Suppose also $\models \langle D|\{p\}\,A\{q\}\rangle$. Now by Lemma 7.4 there is a $p' \simeq wp[\![A, q, D]\!]$ for which $\vdash \langle D|\{p'\}\,A\{q\}\rangle$ in $\mathcal{H}_N$. By Lemma 7.3 and $\models \langle D|\{p\}\,A\{q\}\rangle$ we have $\models p \supset p'$. So, using (A3) and (R1) we get $\vdash \langle D|\{p\}\,A\{q\}\rangle$ in $\mathcal{H}_N$. □

Note that up till now no claims have been made on the interpretation $\mathcal{I}$. The only additional condition was that $\langle D, A\rangle$ should be normal. It can be seen that this is necessary from the fact that

$$\models \langle L\!:\!p|\{\textbf{true}\}\ \textbf{if true then}\ x:=0\ \textbf{else goto}\ L'\ \textbf{fi}\ \{x=0\}\rangle$$

holds, even if $L \not\equiv L'$. However, there is no way to derive this correctness formula in $\mathcal{H}$.

We now turn our attention to the discussion of completeness with respect to correctness formulae of the form $\{p\}\,S\{q\}$. If we want to formally prove such a formula, we have to find suitable label invariants for all labels declared in S. It is at this point that we have to put the restriction on $\mathcal{I}$ which we mentioned in the introduction of this chapter.

The question arises whether in our case such a restriction is necessary. Wand [18] proved that such a restriction is needed for programs without **goto** statements, but containing **while** statements. He constructed an interpretation $\mathcal{I}$ and a correctness formula which was valid under $\mathcal{I}$ but not derivable, because there was no assertion available which could express a suitable invariant. His counterexample can be transferred to our language in such a way that the same arguments he uses can be applied in our situation. Therefore we must make a restriction on $\mathcal{I}$.

Before we can give an exact definition of this restriction *(expressiveness)*, we have to make a few preparations.

Lemma 7.6. *Let $S \equiv [L_i : A_i]_{i=1}^n \in \mathcal{P}rog$ and $\gamma \in \Gamma$. Let ϕ_i be derived from S and γ as in the definition of $\mathcal{M}$. If S is normal, then all ϕ_i are independent of γ.*

Proof. The following holds: Let $A \in \mathcal{S}tat$ and $\{L_1, \ldots, L_n\}$ be the set of all labels occurring in A. Let $\phi, \psi_1, \ldots, \psi_n \in M$ and $\gamma \in \Gamma$. Then $\mathcal{N}[\![A]\!](\gamma\{\psi_i/L_i\}_{i=1}^n)\phi$ is independent of γ. This fact can be proved by induction on the structure of A.

From this result we can infer that the operator

$$\Phi = \lambda\langle\psi_1, \ldots, \psi_n\rangle \cdot \langle \mathcal{N}[\![A_i]\!](\gamma\{\psi_j/L_j\}_{j=1}^n)\psi_{i+1}\rangle_{j=1}^n$$

is independent of γ, and thus the same must be true of $\langle\phi_1, \ldots, \phi_n\rangle$, being the least fixed point $\mu\Phi$ of Φ. □

Definition (transformations derived from S). Let S be a normal program. Then the ϕ_i defined as in Lemma 7.6, are called the *transformations derived from S.*

Definition (weakest precondition of a transformation from M). Let $\phi \in M$, $p \in \mathcal{Ass}$. We say that *q expresses the weakest precondition of ϕ with respect to p* iff $\forall \sigma \in \Sigma_0$: $[\mathcal{T}[\![q]\!]\,\sigma = tt \Leftrightarrow (\phi\sigma \neq \perp \Rightarrow \mathcal{T}[\![p]\!](\phi\sigma) = tt)]$.

Definition (expressiveness). Let $\mathcal{I}$ be an interpretation of the primitive relation and function symbols. We say that $\mathcal{Ass}$ is *expressive relative to $\mathcal{Prog}$ and $\mathcal{I}$*, iff for all assertions p and for all normal programs S the following holds: there are assertions $p_1, \dots, p_n$ such that p_i expresses the weakest precondition of ϕ_i with respect to p, where ϕ_i are the transformations derived from S $(i = 1, \dots, n)$.

If the primitive relation and function symbols of our language $\mathcal{Prog}$ are such that $\mathcal{Ass}$ is a language for Peano arithmetic, and if $\mathcal{I}_0$ is the standard interpretation of this language in the natural numbers then, using recursion theory, we can show that the transformations ϕ_i derived from a normal program S are partial recursive functions in the free variables of S. A result of recursion theory is that for every partial recursive function ϕ: $\mathbb{N}^k \to \mathbb{N}^k$, there is a formula p in $\mathcal{Ass}$ with free variables $x_1, \dots, x_k$, $y_1, \dots, y_k$, which expresses this function, i.e. $\models p(\bar{\alpha}_1, \dots, \bar{\alpha}_k, \bar{\beta}_1, \dots, \bar{\beta}_k)$ iff $\phi(\alpha_1, \dots, \alpha_k)$ is defined and equal to $\langle \beta_1, \dots, \beta_k \rangle$ (where $\bar{\alpha}_i, \bar{\beta}_i$ are numerals denoting the natural numbers α_i, β_i). From this we can infer that $\mathcal{Ass}$ is expressive relative to $\mathcal{Prog}$ and the standard interpretation $\mathcal{I}_0$.

Now we have enough tools to state the main lemma needed to prove completeness for formulae of the form $\{p\}\,S\,\{q\}$. In essence this theorem states that the p_i from the definition of expressiveness are the label invariants which we are looking for.

Lemma 7.7. *Let $\mathcal{I}$ be an interpretation such that $\mathcal{Ass}$ is expressive relative to $\mathcal{Prog}$ and $\mathcal{I}$. Let $S \equiv [L_i : A_i]_{i=1}^n \in \mathcal{Prog}$ be normal. Let $p \in \mathcal{Ass}$, and let ϕ_i be the transformations derived from S $(i = 1, \dots, n)$. Let p_i be the assertions expressing the weakest preconditions of ϕ_i with respect to p $(i = 1, \dots, n)$, and let $p_{n+1} \equiv p$. Then*

$$\models \langle [L_i : p_i]_{i=1}^n \mid \{p_j\}\, A_j\, \{p_{j+1}\} \rangle$$

for $j = 1, \dots, n$.

Proof. Choose $j (1 \leqq j \leqq n)$ and $\sigma \in \Sigma_0$ such that $\mathcal{T}[\![p_j]\!]\,\sigma = tt$. There are two cases:

a) $\mathcal{A}[\![A_j]\!]\,\sigma = \sigma' \in \Sigma_0$. According to the definition of $\models \langle D \mid \{p_j\}\, A_j\, \{p_{j+1}\} \rangle$, in this case we have to prove that $\mathcal{T}[\![p_{j+1}]\!]\,\sigma' = tt$. Now by the assumption on p_j and by definition of weakest precondition we have $\mathcal{T}[\![p_j]\!]\,\sigma = tt \Leftrightarrow (\phi_j\sigma \neq \perp \Rightarrow \mathcal{T}[\![p]\!](\phi_j\sigma) = tt)$. Also, $\phi_j\sigma = \phi_{j+1}\sigma'$ (by 4.3.1°, $\mathcal{A}[\![A_j]\!]\,\sigma = \sigma'$ and $\phi_j\sigma = \mathcal{N}[\![A_j]\!](\gamma\{\phi_i/L_i\}_{i=1}^n)\,\phi_{j+1}\sigma)$. Combining these results, we get $\mathcal{T}[\![p_j]\!]\,\sigma = tt \Leftrightarrow (\phi_{j+1}\sigma' \neq \perp \Rightarrow \mathcal{T}[\![p]\!](\phi_{j+1}\sigma') = tt)$. But the right-hand side of this equivalence is (by definition of weakest precondition, and by the assumption on p_{j+1}) equivalent to $\mathcal{T}[\![p_{j+1}]\!]\,p' = tt$.

b) $\mathcal{A}[\![A_j]\!]\,\sigma = \langle \sigma', L \rangle \in \Sigma_0 \times \mathcal{Lvar}$. From the fact that S is normal, we infer that L must be some L_k $(1 \leqq k \leqq n)$. We have to prove (by definition of $\models \langle D \mid \{p\}\, A\, \{q\} \rangle$) that $\mathcal{T}[\![p_k]\!]\,\sigma' = tt$. Again we have:

$\mathcal{T}[\![p_j]\!]\,\sigma = tt \Leftrightarrow (\phi_j\sigma \neq \perp \Rightarrow \mathcal{T}[\![p]\!](\phi_j\sigma) = tt)$, and now we have $\phi_j\sigma = \phi_k\sigma'$ by 4.3.2° (analogously to a)). Combining the results, we get

$$\begin{aligned}\mathscr{T}[\![p_j]\!]\,\sigma = tt &\Leftrightarrow (\phi_k\sigma' \neq \bot \Rightarrow \mathscr{T}[\![p]\!]\,(\phi_k\sigma') = tt)\\ &\Leftrightarrow \mathscr{T}[\![p_k]\!]\,\sigma' = tt. \quad \square\end{aligned}$$

We now can collect our results in the completeness theorem 7.8.

Theorem 7.8. *The deduction system $\mathscr{H}_N$ is complete in the sense of Cook, i.e., for every interpretation $\mathscr{I}$ such that $\mathscr{A}\!ssn$ is expressive relative to $\mathscr{P}\!rog$ and $\mathscr{I}$, and for every normal correctness formula f, we have $\models f \Rightarrow \vdash f$ in $\mathscr{H}_N$.*

Proof. a) If $f \equiv p \in \mathscr{A}\!ssn$, then $\models p \Rightarrow \vdash p$ by (A 3)

b) If $f \equiv \langle D \,|\, \{p\}\, A\, \{q\}\rangle$ then we can apply Lemma 7.5.

c) $f \equiv \{p\}\, S\, \{q\}$. Say $S \equiv [L_i : A_i]_{i=1}^n$. Let ϕ_i be the transformations derived from S. By Lemma 7.7 there are assertions p_j, expressing the weakest preconditions of ϕ_j with respect to q such that

$$\models \langle [L_i : p_i]_{i=1}^n \,|\, \{p_j\}\, A_j\, \{p_{j+1}\}\rangle$$

for $j = 1, \dots, n$, where $p_{n+1} \equiv p$.

Note that these correctness formulae are normal by the fact that $\{p\}\, S\, \{q\}$ and thus S is normal. Lemma 7.5 then gives us

$$\vdash \langle [L_i : p_i]_{i=1}^n \,|\, \{p_j\}\, A_j\, \{p_{j+1}\}\rangle \quad \text{in } \mathscr{H}_N$$

for $j = 1, \dots, n$. Now we can apply rule (R 5) to get

$$\vdash \{p_1\}\, S\, \{p_{n+1}\} \quad \text{in } \mathscr{H}_N.$$

Now $p_{n+1} \equiv q$. Moreover $\models p \supset p_1$. For, assume $\mathscr{T}[\![p]\!]\,\sigma = tt$ for some $\sigma \in \Sigma_0$. Then, by $\models \{p\}\, S\, \{q\}$, we have $\forall \gamma \in \Gamma\ \forall \sigma' \in \Sigma_0 \colon \sigma' = \mathscr{M}[\![S]\!]\,\gamma\sigma \Rightarrow \mathscr{T}[\![q]\!]\,\sigma' = tt$. But $\mathscr{M}[\![S]\!]\,\gamma\sigma = \phi_1\sigma$ (Lemma 4.2.2). Thus: $\sigma' = \phi_1\sigma \in \Sigma_0 \Rightarrow \mathscr{T}[\![q]\!]\,\sigma' = tt$. But this is equivalent to $\mathscr{T}[\![p_1]\!]\,\sigma = tt$, using the definition of weakest precondition.

We had $\vdash \{p_1\}\, S\, \{q\}$. Also $\models p \supset p_1$, and thus $\vdash p \supset p_1$ by (A 3). Finally, using (R 2), we conclude $\vdash \{p\}\, S\, \{q\}$ in $\mathscr{H}_N$. $\square$

8. Deduction System: Second Variant

The validity definition of Sect. 6 makes explicit use of the label invariants p_i, which therefore had to be provided by the formulae of the deduction system. The purpose of this chapter is to show that it is possible to define validity in such a way that the label invariants are not explicitly needed. We will, using continuation semantics, associate a truth value with a formula $\{p\}\, A\, \{q\}$. This truth value will be dependent on the meaning of the labels occurring within A, i.e. the value depends on the environment γ. Consequently, we will establish a semantical function $\mathscr{G}$ such that for every A, p and q we have $\mathscr{G}[\![\{p\}\, A\, \{q\}]\!]\colon \Gamma \to \{ff, tt\}$.

This leads to a definition of validity which turns out to be equivalent to the one of Sect. 6 in the following sense: $\langle [L_i : p_i]_{i=1}^n \,|\, \{p\}\, A\, \{q\}\rangle$ is valid according to the definition in Sect. 6, if and only if $\{p\}\, A\, \{q\}$ is valid (using function $\mathscr{G}$) in every environment γ for which all formulae $\{p_i\}$ **goto** L_i $\{$**false**$\}$ are valid ($i=1,\ldots,n$). Or more formally,

$$\models \langle [L_i : p_i]_{i=1}^n \mid \{p\}\, A\, \{q\}\rangle$$
$$\Leftrightarrow \forall \gamma \in \Gamma \left[\bigwedge_{i=1}^{n} (\mathscr{G}[\![\{p_i\}\ \textbf{goto}\ L_i\ \{\textbf{false}\}]\!]\gamma = tt) \Rightarrow \mathscr{G}[\![\{p\}\, A\, \{q\}]\!]\gamma = tt \right].$$

Using this new approach we can define validity for the system as given in [8]. But, before we do that, we change this system somewhat. The system in [8] is presented as a natural deduction system, which means that the notion "proof from assumptions" is used. A line in a formal proof can be a formula which is introduced as an assumption. The system also has an inference rule in which assumptions are discharged, namely

$$\frac{\begin{array}{l}\{p'\}\ \textbf{goto}\ L\ \{\textbf{false}\} \vdash \{p\}\, A_1\, \{p'\}\\ \{p'\}\ \textbf{goto}\ L\ \{\textbf{false}\} \vdash \{p'\}\, A_2\, \{q\}\end{array}}{\{p\}\, A_1 ;\, L\colon A_2\, \{q\}}$$

which discharges the assumption $\{p'\}$ **goto** L $\{$**false**$\}$, needed in the derivation of $\{p\}\, A_1\, \{p'\}$ and $\{p'\}\, A_2\, \{q\}$. Thus, every derived formula f in the system of [8] will have a finite set Δ of assumptions attached by it, namely those assumptions which were used to derive f.

We transform this natural deduction system into a sequent calculus having formulae of the form

$$\Delta \to \{p\}\, A\, \{q\},$$

where Δ is meant to be the finite set of assumptions associated with the derivation of $\{p\}\, A\, \{q\}$. The advantage of this system over the natural deduction system is that validity of a formula can be defined more directly, now that every formula incorporates the relevant assumptions.

We now define the deduction system

Definition (atomic correctness formula, correctness formula). An *atomic correctness formula* is a formula of the form $\{p\}\, A\, \{q\}$. The class of all atomic correctness formulae will be denoted by $\mathscr{A}\!\mathit{for}$, and has g as a typical element.

A *correctness formula* is either an assertion, or a formula of the form $\Delta \to \{p\}\, A\, \{q\}$, or a formula of the form $\Delta \to \{p\}\, S\, \{q\}$, where Δ (the set of *assumptions*) is a finite set of atomic correctness formulae. The class of all correctness formulae will be denoted by $\mathscr{C}\!\mathit{orr}$, and has f as a typical element.

The correctness formulae $\phi \to \{p\}\, A\, \{q\}$ and $\phi \to \{p\}\, S\, \{q\}$ will be abbreviated to $\{p\}\, A\, \{q\}$ and $\{p\}\, S\, \{q\}$ respectively.

Definition (deduction system $\mathscr{H}'$). The axioms are

(A 1) $\Delta \to \{p[s/x]\}\, x := s\, \{p\}$

(A 2) $\Delta \to g$, where $g \in \Delta$

(A 3) p,

where p is a valid assertion (i.e. $\forall \sigma \in \Sigma_0\colon \mathscr{T}[\![p]\!]\sigma = tt$).

The rules of inference are

(R 1) $$\frac{p_1 \supset p_2, p_3 \supset p_4, \Delta \rightarrow \{p_2\} A \{p_3\}}{\Delta \rightarrow \{p_1\} A \{p_4\}}$$

(R 2) $$\frac{p_1 \supset p_2, p_3 \supset p_4, \Delta \rightarrow \{p_2\} S \{p_3\}}{\Delta \rightarrow \{p_1\} S \{p_4\}}$$

(R 3) $$\frac{\Delta \rightarrow \{p_1\} A \{p_2\}, \Delta \rightarrow \{p_2\} A' \{p_3\}}{\Delta \rightarrow \{p_1\} A; A' \{p_3\}}$$

(R 4) $$\frac{\Delta \rightarrow \{p \wedge b\} A \{q\},\ \Delta \rightarrow \{p \wedge \neg b\} A' \{q\}}{\Delta \rightarrow \{p\}\ \textbf{if}\ b\ \textbf{then}\ A\ \textbf{else}\ A'\ \textbf{fi}\ \{q\}}$$

(R 5) $$\frac{\Delta \cup \Delta' \rightarrow \{p_1\} A_1 \{p_2\}, \ldots, \Delta \cup \Delta' \rightarrow \{p_n\} A_n \{p_{n+1}\}}{\Delta' \rightarrow \{p_1\} [L_i : A_i]_{i=1}^{n} \{p_{n+1}\}}$$

where $\Delta = \{\{p_i\}\ \textbf{goto}\ L_i\ \{\textbf{false}\} \mid i = 1, \ldots, n\}$, all L_i are different, and no L_i occurs in any assumption in Δ' $(i = 1, \ldots, n)$.

The restriction on Δ' in (R 5) is imposed to circumvent possibilities like the following. Suppose $\Delta = \{\{\textbf{true}\}\ \textbf{goto}\ L_1\ \{\textbf{false}\}, \{x=0\}\ \textbf{goto}\ L_2\ \{\textbf{false}\}\}$ and Δ' is the singleton $\{\{\textbf{true}\}\ \textbf{goto}\ L_2\ \{\textbf{false}\}\}$. Then we can derive

$$\Delta \cup \Delta' \rightarrow \{\textbf{true}\}\ x := 1;\ \textbf{goto}\ L_2\ \{x=0\} \qquad \ldots(\#)$$

using the assumption in Δ', and furthermore

$$\Delta \cup \Delta' \rightarrow \{x=0\}\ x := x\ \{x=0\}.$$

Thus, discharging Δ, using "(R 5)"

$$\Delta' \rightarrow \{\textbf{true}\}\ L_1 : x := 1;\ \textbf{goto}\ L_2; L_2 : x := x\ \{x=0\} \qquad \ldots(b)$$

and this formula is not valid (the assumption $\{\textbf{true}\}\ \textbf{goto}\ L_2\ \{\textbf{false}\}$ in Δ' is not relevant for the validity of (b), because the meaning of $L_1 : x := 1;\ \textbf{goto}\ L_2; L_2 : x := x$ in any γ, given by $\mathscr{M}$, doesn't depend on the meaning $\gamma[\![L_2]\!]$ of L_2 anymore). Difficulties stem from the fact that the assumption in Δ' was used in the derivation of $(\#)$, and not discharged.

We now come to the definition of the function $\mathscr{G}$ which we shall need to define validity of correctness formulae. The main problem in defining the value of $\mathscr{G}[\![\{p\} A \{q\}]\!]$ in some environment γ is that $\mathscr{N}[\![A]\!]\gamma\phi$ is not a function that transforms states just before evaluation of A into states immediately after this evaluation, while q is an assertion describing the latter states.

We can however say something about the states at the normal exit point of A in the following indirect way. Consider the formula $\{p\} A \{q\}$ and choose a predicate π (a function in $\Sigma_0 \rightarrow \{ff, tt\}$) which we want to be true in every final state $\sigma' = \mathscr{N}[\![A]\!]\gamma\phi\sigma$ corresponding to an initial state σ satisfying $\mathscr{T}[\![p]\!]\sigma = tt$. That is, we want

$$\forall \sigma, \sigma' \in \Sigma_0 : [(\mathscr{T}[\![p]\!]\sigma = tt \wedge \sigma' = \mathscr{N}[\![A]\!]\gamma\phi\sigma) \Rightarrow \pi\sigma' = tt].$$

We will abbreviate this partial correctness condition to

$$\{\mathcal{T}[\![p]\!]\}\,\mathcal{N}[\![A]\!]\,\gamma\phi\,\{\pi\}.$$

Now, as this formula must correspond to $\{p\}\,A\,\{q\}$, we are looking for a relation between q, the continuation ϕ chosen, and the predicate π. It is reasonable to demand that $\pi(\phi\,\sigma'')=tt$ for every (intermediate) state σ'' satisfying $\mathcal{T}[\![q]\!]\,\sigma''=tt$ (provided $\phi\sigma''\neq\bot$). For, the continuation ϕ is the state transformation describing what happens after evaluation of A has terminated at the normal exit point. So we want q, ϕ and π to be related through $\{\mathcal{T}[\![q]\!]\}\,\phi\,\{\pi\}$. It turns out that this constraint on ϕ and π is sufficient to lead to a satisfying validity definition.

Definition (predicates; partial correctness, semantical level). The class of *predicates* Π, with typical element π, is defined by $\Pi=\Sigma_0\to\{ff,tt\}$. For any $\pi,\pi'\in\Pi$ and $\phi\in M$, we define

$$\{\pi\}\,\phi\,\{\pi'\}\Leftrightarrow\forall\,\sigma,\sigma'\in\Sigma_0\colon[(\pi\sigma=tt\wedge\sigma'=\phi\sigma)\Rightarrow\pi'\sigma'=tt].$$

Definition ($\mathcal{G}$). The function $\mathcal{G}$ with functionality $\mathcal{G}\colon\mathcal{A}for\to\Gamma\to\{ff,tt\}$ is defined by

$$\mathcal{G}[\![\{p\}\,A\,\{q\}]\!]\,\gamma=tt\Leftrightarrow\forall\,\pi\in\Pi\;\forall\,\phi\in M\,[\{\mathcal{T}[\![q]\!]\}\,\phi\,\{\pi\}\Rightarrow\{\mathcal{T}[\![p]\!]\}\,\mathcal{N}[\![A]\!]\,\gamma\phi\,\{\pi\}].$$

We extend the domain of $\mathcal{G}$ to subsets Δ of $\mathcal{A}for$ as follows

$$\mathcal{G}[\![\Delta]\!]\,\gamma=tt\Leftrightarrow\forall f\in\Delta\colon\mathcal{G}[\![f]\!]\,\gamma=tt.$$

Definition (validity). A correctness formula f is *valid* (written $\models f$) is

1°. $f\equiv p$ and $\forall\,\sigma\in\Sigma_0\colon\mathcal{T}[\![p]\!]\,\sigma=tt$, or

2°. $f\equiv\Delta\to\{p\}\,A\,\{q\}$ and $\forall\,\gamma\in\Gamma\colon\mathcal{G}[\![\Delta]\!]\,\gamma=tt\Rightarrow\mathcal{G}[\![\{p\}\,A\,\{q\}]\!]\,\gamma=tt$, or

3°. $f\equiv\Delta\to\{p\}\,S\,\{q\}$ and $\forall\,\gamma\in\Gamma\colon\mathcal{G}[\![\Delta]\!]\,\gamma=tt\Rightarrow\{\mathcal{T}[\![p]\!]\}\,\mathcal{M}[\![S]\!]\,\gamma\,\{\mathcal{T}[\![q]\!]\}$.

We now investigate whether the system as it stands now is sound and complete. It will be proven at the end of this chapter that the system is sound. However, the system is not complete. For instance, a formula like

$$\{\{p\}\,x:=x;\ \textbf{goto}\ L\{q\}\}\to\{p\}\ \textbf{goto}\ L\{q\}$$

is valid but not derivable. Therefore we first prove soundness and completeness of a restriction of the system, namely the system consisting of normal correctness formulae only.

Definition (normal correctness formulae). A correctness formula f is called *normal* if

1°. $f\equiv p$, or

2°. $f\equiv\Delta\to\{p\}\,A\,\{q\}$, where $\Delta=\{\{p_i\}\,\textbf{goto}\ L_i\,\{\textbf{false}\}\mid i=1,\dots,n\}$ such that all L_i are different and that the labels in A are all L_i's, or

3°. $f\equiv\{p\}\,S\,\{q\}$, where S is a normal program.

The system $\mathcal{H}'$, restricted to the normal formulae, is called *the normal fragment of* $\mathcal{H}'$, and denoted by $\mathcal{H}'_N$.

There is an obvious one to one correspondence between the normal correctness formulae as defined here and the normal correctness formulae from Sect. 6, given by the function Φ, defined by

$$\Phi[\![p]\!]=p$$
$$\Phi[\![\{\{p_i\}\ \mathbf{goto}\ L_i\ \{\mathbf{false}\}\mid i=1,\dots,n\}\rightarrow\{p\}\,A\{q\}]\!]=\langle[L_i\colon p_i]_{i=1}^{n}\mid\{p\}\,A\{q\}\rangle$$
$$\Phi[\![\{p\}\,S\{q\}]\!]=\{p\}\,S\{q\}.$$

If we compare the axioms and inference rules of $\mathscr{H}$ with the ones of $\mathscr{H}'$ we come to the following lemma:

Lemma 8.1. *For every normal correctness formula f we have*

$$\vdash f\,(\text{in } \mathscr{H}')\Leftrightarrow\ \vdash \Phi[\![\mathrm{f}]\!]\,(\text{in } \mathscr{H}).$$

Proof. The $\Leftarrow-$ direction is obvious. The proof of "$\Rightarrow$" essentially amounts to showing that $\mathscr{H}'$ is *conservative over* $\mathscr{H}'_N$, i.e. if a normal formula is derivable in $\mathscr{H}'$ then it has a proof in $\mathscr{H}'_N$. This can be shown using the fact that every inference rule has normal premisses if its conclusion is a normal formula. □

If we can prove the same result for validity instead of deducibility then we can infer from the results in Sect. 7 that $\mathscr{H}'_N$ is sound and complete. To achieve this, we first prove some lemmas, relating the definition of validity of $f\equiv\Delta\rightarrow\{p\}\,A\{q\}$ with validity of $\Phi[\![\mathrm{f}]\!]$.

Lemma 8.2. *Suppose $f\equiv\{\{p_i\}$ **goto** L_i $\{$**false**$\}\mid i=1,\dots,n\}\rightarrow\{p\}\,A\{q\}$ is a correctness formula that is normal and valid. Then the following holds:*

a) $\forall\sigma,\sigma'\in\Sigma_0\colon(\mathscr{A}[\![A]\!]\,\sigma=\sigma'\wedge\mathscr{T}[\![p]\!]\,\sigma=tt)\Rightarrow\mathscr{T}[\![q]\!]\,\sigma'=tt$

b) $\forall\sigma,\sigma'\in\Sigma_0\colon(\mathscr{A}[\![A]\!]\,\sigma=\langle\sigma',L_i\rangle\wedge\mathscr{T}[\![p]\!]\,\sigma=tt)\Rightarrow\mathscr{T}[\![p_i]\!]\,\sigma'=tt\ (i=1,\dots,n).$

Proof. a) Choose $\sigma,\sigma'\in\Sigma_0$ such that $\mathscr{T}[\![p]\!]\,\sigma=tt$ and $\mathscr{A}[\![A]\!]\,\sigma=\sigma'$. Choose γ_0 such that $\gamma_0[\![L_i]\!]=\lambda\sigma\cdot\bot$ for $i=1,\dots,n$. Then we can check that $\mathscr{G}[\![\{p_i\}$ **goto** L_i $\{$**false**$\}]\!]\,\gamma_0=tt$ for $i=1,\dots,n$ and thus from validity of f we get $\mathscr{G}[\![\{p\}\,A\{q\}]\!]\,\gamma_0=tt$. From the definition of $\mathscr{G}$ we then have

$$\forall\pi\in\Pi\ \forall\phi\in M[\{\mathscr{T}[\![q]\!]\}\,\phi\{\pi\}\Rightarrow\{\mathscr{T}[\![p]\!]\}(\mathscr{N}[\![A]\!]\,\gamma_0\phi)\{\pi\}].$$

If we choose $\pi=\mathscr{T}[\![q]\!]$ and $\phi=\lambda\sigma\cdot\sigma$, we can deduce from this

$$\{\mathscr{T}[\![p]\!]\}(\mathscr{N}[\![A]\!]\,\gamma_0\{\lambda\sigma\cdot\sigma\})\{\mathscr{T}[\![q]\!]\}.$$

Combining this with $\mathscr{N}[\![A]\!]\,\gamma_0\{\lambda\sigma\cdot\sigma\}\sigma=\sigma'$ (Lemma 4.3.1°) and $\mathscr{T}[\![p]\!]\,\sigma=tt$, we get $\mathscr{T}[\![q]\!]\,\sigma'=tt$.

b) Choose $\sigma,\sigma'\in\Sigma_0$ and i (where $1\leqq i\leqq n$) such that $\mathscr{T}[\![p]\!]\,\sigma=tt$ and $\mathscr{A}[\![A]\!]\,\sigma=\langle\sigma',L_i\rangle$. Now, if we take γ_0 such that

$$\gamma_0[\![L_j]\!]\,\sigma=\begin{cases}\sigma & \text{if } \sigma\neq\bot \text{ and } \mathscr{T}[\![p_j]\!]\,\sigma=ff\\ \bot & \text{otherwise}\end{cases}$$

for $j=1,\ldots,n$, we again have that $\mathscr{G}[\![\{p_j\}\ \mathbf{goto}\ L_j\ \{\mathbf{false}\}]\!]\gamma_0=tt$ $(j=1,\ldots,n)$. Arguing the same way as in the proof of a) we come to

$$\forall\pi\in\Pi\ \forall\phi\in M[\{\mathscr{T}[\![q]\!]\}\ \phi\{\pi\}\Rightarrow\{\mathscr{T}[\![p]\!]\}(\mathscr{N}[\![A]\!]\gamma_0\phi)\{\pi\}].$$

Now we choose $\phi=\lambda\sigma\cdot\bot$ and $\pi=\lambda\sigma\cdot ff$. We then derive

$$\{\mathscr{T}[\![p]\!]\}(\mathscr{N}[\![A]\!]\gamma_0\{\lambda\sigma\cdot\bot\})\{\lambda\sigma\cdot ff\}.$$

Combining this with $\mathscr{N}[\![A]\!]\gamma_0\{\lambda\sigma\cdot\bot\}\sigma=\gamma_0[\![L_i]\!]\sigma'$ (Lemma 4.3.2°) and with $\mathscr{T}[\![p]\!]\sigma=tt$ we have that

$$\gamma_0[\![L_i]\!]\sigma'\neq\bot\Rightarrow(\lambda\sigma\cdot ff)(\gamma_0[\![L_i]\!]\sigma')=tt.$$

So we must have $\gamma_0[\![L_i]\!]\sigma'=\bot$, but this is (by definition of γ_0 and the fact that $\sigma'\neq\bot$) equivalent to $\mathscr{T}[\![p_i]\!]\sigma'=tt$. □

Lemma 8.3. *Suppose* $f\equiv\{\{p_i\}\ \mathbf{goto}\ L_i\ \{\mathbf{false}\}\mid i=1,\ldots,n\}\rightarrow\{p\}\ A\{q\}$ *is a normal correctness formula. Then*

$$\models f\Leftrightarrow\forall\sigma\in\Sigma_0:\mathscr{T}[\![p]\!]\sigma=tt\Rightarrow\begin{bmatrix}(\exists\sigma'\in\Sigma_0[\mathscr{A}[\![A]\!]\sigma=\sigma'\wedge\mathscr{T}[\![q]\!]\sigma'=tt])\vee\\(\exists\sigma'\in\Sigma_0[\mathscr{A}[\![A]\!]\sigma=\langle\sigma',L_i\rangle\wedge\mathscr{T}[\![p_i]\!]\sigma'=tt])\end{bmatrix}.$$

Proof. "⇒". Suppose $\models f$ and $\mathscr{T}[\![p]\!]\sigma=tt$. There are two possibilities (by definition of $\mathscr{A}$)

a) $\mathscr{A}[\![A]\!]\sigma=\sigma'\in\Sigma_0$, and Lemma 8.2a yields $\mathscr{T}[\![q]\!]\sigma'=tt$

b) $\mathscr{A}[\![A]\!]\sigma=\langle\sigma',L\rangle$. Since f is normal, which means that all labels in A are an L_i, we have that L is an L_i for some i $(1\leqq i\leqq n)$. We then can apply Lemma 8.2b to obtain $\mathscr{T}[\![p_i]\!]\sigma'=tt$.

"⇐". Choose $\gamma\in\Gamma$ such that $\mathscr{G}[\![\{p_i\}\ \mathbf{goto}\ L_i\ \{\mathbf{false}\}]\!]\gamma=tt$ for $i=1,\ldots,n$. Then we must derive $\mathscr{G}[\![\{p\}\ A\{q\}]\!]\gamma=tt$, or equivalently

$$\forall\pi\in\Pi\ \forall\phi\in M[\{\mathscr{T}[\![q]\!]\}\ \phi\{\pi\}\Rightarrow\{\mathscr{T}[\![p]\!]\}(\mathscr{N}[\![A]\!]\gamma\phi)\{\pi\}].$$

So choose π_0 and ϕ_0 such that $\{\mathscr{T}[\![q]\!]\}\ \phi_0\{\pi_0\}$ holds, and choose σ such that $\mathscr{T}[\![p]\!]\sigma=tt$. We have to prove $\sigma''=\mathscr{N}[\![A]\!]\gamma\phi_0\sigma\neq\bot\Rightarrow\pi_0\sigma''=tt$. Again we have two possibilities:

a) $\mathscr{A}[\![A]\!]\sigma=\sigma'$. Then by assumption $\mathscr{T}[\![q]\!]\sigma'=tt$, and by Lemma 4.3.1°: $\sigma''=\mathscr{N}[\![A]\!]\gamma\phi_0\sigma=\phi_0\sigma'$. From $\{\mathscr{T}[\![q]\!]\}\ \phi_0\{\pi_0\}$ we then have $\sigma''\neq\bot\Rightarrow\pi_0\sigma''=tt$.

b) $\mathscr{A}[\![A]\!]\sigma=\langle\sigma',L_i\rangle$. By assumption $\mathscr{T}[\![p_i]\!]\sigma'=tt$, and by Lemma 4.3.2°: $\sigma''=\mathscr{N}[\![A]\!]\gamma\phi_0\sigma=\gamma[\![L_i]\!]\sigma'$. Now we use the fact that $\mathscr{G}[\![\{p_i\}\ \mathbf{goto}\ L_i\ \{\mathbf{false}\}]\!]\gamma=tt$, or $\forall\pi\in\Pi\ \forall\phi\in M[\{\mathscr{T}[\![\mathbf{false}]\!]\}\ \phi\{\pi\}\Rightarrow\{\mathscr{T}[\![p_i]\!]\}(\gamma[\![L_i]\!])\{\pi\}]$. Taking $\pi=\pi_0$ and $\phi=\lambda\sigma\cdot\bot$, we get $\{\mathscr{T}[\![p_i]\!]\}(\gamma[\![L_i]\!])\{\pi_0\}$, and from this we prove $\sigma''\neq\bot\Rightarrow\pi_0\sigma''=tt$. □

Corollary. *For all normal correctness formulae f we have*

$\models f$ *(according to the validity definition of this chapter)* ⇔

$\models\Phi[\![f]\!]$ *(according to the definition of Sect. 6).*

Proof. This is the lemma for $f \equiv \Delta \to \{p\}\, A\{q\}$. For all other cases for f we have that the respective validity definitions are the same. □

This corollary and the results of Sect. 7 now lead to

Theorem 8.4. *The system $\mathscr{H}'_N$ is sound and complete in the sense of Cook.*

To conclude this section we show that system $\mathscr{H}'$ is sound although, as we have seen before, not complete.

Theorem 8.5. *For all correctness formulae f, we have $\vdash f \Rightarrow \models f$.*

Proof. The proof is analogous to the proof of 7.1. We prove here the more interesting cases, viz. validity of (A1), (R1) and (R5).

(A1) Validity is proven if we can show that

$$\forall \gamma \in \Gamma\ \forall \phi \in M\ \forall \pi \in \Pi [\{\mathscr{T}[\![p]\!]\}\, \phi\{\pi\} \Rightarrow \{\mathscr{T}[\![p[s/x]]\!]\}(\mathscr{N}[\![x := s]\!]\,\gamma\phi)\{\pi\}.$$

So, choose γ, ϕ, π and σ such that $\{\mathscr{T}[\![p]\!]\}\,\phi\{\pi\}$ and $\mathscr{T}[\![p[s/x]]\!]\,\sigma = tt$ hold. Lemma 6.1d then gives $\mathscr{T}[\![p]\!]\,\sigma' = tt$, where $\sigma' = \sigma\{\mathscr{V}[\![s]\!]\,\sigma/x\}$. Now from $\mathscr{N}[\![x := s]\!]\,\gamma\phi\sigma = \phi\sigma'$ and $\{\mathscr{T}[\![p]\!]\}\,\phi\{\pi\}$ we have $\sigma'' = \mathscr{N}[\![x := s]\!]\,\gamma\phi\sigma = \phi\sigma' \neq \bot$ $\Rightarrow \pi\sigma'' = tt$.

(R1) Suppose $\models p_1 \supset p_2$, $\models p_3 \supset p_4$ and $\models \Delta \to \{p_2\}\, A\{p_3\}$. We then have to prove $\models \Delta \to \{p_1\}\, A\{p_4\}$. Suppose that we have a $\gamma \in \Gamma$ such that $\mathscr{G}[\![\Delta]\!]\,\gamma = tt$ (if there is no such γ then $\Delta \to \{p_1\}\, A\{p_4\}$ is vacuously valid). We then must prove $\mathscr{G}[\![\{p_1\}\, A\{p_4\}]\!]\,\gamma = tt$, or $\forall \phi \in M\, \forall \pi \in \Pi [\{\mathscr{T}[\![p_4]\!]\}\,\phi\{\pi\} \Rightarrow \{\mathscr{T}[\![p_1]\!]\}\,\mathscr{N}[\![A]\!]\,\gamma\phi\{\pi\}]$. We will prove this using the following fact:

$$\text{if } \forall \sigma \in \Sigma_0\colon \pi\sigma = tt \Rightarrow \pi'\sigma = tt, \quad \text{then } \{\pi'\}\,\phi\{\bar{\pi}\} \Rightarrow \{\pi\}\,\phi\{\bar{\pi}\} \qquad \ldots(*)$$

which can easily be verified.

Now suppose $\{\mathscr{T}[\![p_4]\!]\}\,\phi\{\pi\}$. From $\models p_3 \supset p_4$ and $(*)$ we get $\{\mathscr{T}[\![p_3]\!]\}\,\phi\{\pi\}$. From this, $\models \Delta \to \{p_2\}\, A\{p_3\}$ and $\mathscr{G}[\![\Delta]\!]\,\gamma = tt$, we have $\{\mathscr{T}[\![p_2]\!]\}(\mathscr{N}[\![A]\!]\,\gamma\phi)\{\pi\}$. Then we use $\models p_1 \supset p_2$ and $(*)$ again to derive $\{\mathscr{T}[\![p_1]\!]\}(\mathscr{N}[\![A]\!]\,\gamma\phi)\{\pi\}$.

(R5) Let $\Delta = \{\{p_i\}\ \textbf{goto}\ L_i\ \{\textbf{false}\} \mid i = 1, \ldots, n\}$ where all L_i are different, and let Δ' be such that no L_i occurs in any formula in Δ'. Suppose furthermore that we have $\models \Delta \cup \Delta' \to \{p_i\}\, A_i\{p_{i+1}\}$ for $i = 1, \ldots, n$. We have to prove $\models \Delta' \to \{p_1\}\,[L_i\colon A_i]_{i=1}^n\,\{p_{n+1}\}$.

So, choose γ such that $\mathscr{G}[\![\Delta']\!]\,\gamma = tt$. Let ϕ_i, $\phi_i^{(k)}$ and $\gamma^{(k)}$ $(i = 1, \ldots, n;\ k = 0, 1, \ldots)$ be derived from γ and $[L_i\colon A_i]_{i=1}^n$ as in Lemma 4.2. We then have to prove $\{\mathscr{T}[\![p_1]\!]\}\,\phi_1\{\mathscr{T}[\![p_{n+1}]\!]\}$ (take $i = 1$ in Lemma 4.2.2). Now if we can prove

$$\forall k \{\mathscr{T}[\![p_i]\!]\}\,\phi_i^{(k)}\{\mathscr{T}[\![p_{n+1}]\!]\} \qquad (i = 1, \ldots, n+1) \qquad \ldots(\#)$$

then we are ready. For suppose we have $\sigma, \sigma' \in \Sigma_0$ such that $\mathscr{T}[\![p_1]\!]\,\sigma = tt$ and $\sigma' = \phi_1\sigma$. Then (since $\phi_1 = \bigsqcup_k (\phi_1^{(k)})$ we have $\sigma' = (\bigsqcup_k \phi_1^{(k)})\,\sigma = \bigsqcup_k (\phi_1^{(k)}\sigma)$, and because Σ is a discrete cpo there must be a $\bar{k}$ such that $\phi_1^{(\bar{k})}\sigma = \sigma'$. But then we can infer that $\mathscr{T}[\![p_{n+1}]\!]\,\sigma' = tt$ by applying $(\#)$ with $i = 1$ and $k = \bar{k}$.

We now prove $(\#)$ by induction on k. The basis $(k = 0)$ is trivial, so we now perform the induction step. Choose i $(1 \leqq i \leqq n$, the case $i = n+1$ being trivial).

We have to prove $\{\mathscr{T}[\![p_i]\!]\}\,\phi_i^{(k+1)}\{\mathscr{T}[\![p_{n+1}]\!]\}$ or $\{\mathscr{T}[\![p_i]\!]\}$ $(\mathscr{N}[\![A_i]\!]\,\gamma^{(k)}\phi_{i+1}^{(k)})$ $\{\mathscr{T}[\![p_{n+1}]\!]\}$.

Choose $\sigma, \sigma'' \in \Sigma_0$ such that $\mathscr{T}[\![p_i]\!]\,\sigma = tt$ and $\sigma'' = \mathscr{N}[\![A_i]\!]\,\gamma^{(k)}\phi_{i+1}^{(k)}\,\sigma$. We have to show $\mathscr{T}[\![p_{n+1}]\!]\sigma'' = tt$. We do this in a way that is analogous to the proof of Lemma 8.2, i.e. by choosing a suitable environment γ_0. We distinguish three cases: $\mathscr{A}[\![A_i]\!]\,\sigma = \sigma'$, $\mathscr{A}[\![A_i]\!]\,\sigma = \langle\sigma', L_j\rangle$ and $\mathscr{A}[\![A_i]\!]\,\sigma = \langle\sigma', L\rangle$, where L is not an L_j.

1°. $\mathscr{A}[\![A_i]\!]\,\sigma = \sigma'$ and thus (4.3.1°) $\sigma'' = \mathscr{N}[\![A_i]\!]\,\gamma^{(k)}\phi_{i+1}^{(k)}\,\sigma = \phi_{i+1}^{(k)}\,\sigma'$. So, if we prove $\mathscr{T}[\![p_{i+1}]\!]\,\sigma' = tt$, we can use the induction hypothesis to infer that $\mathscr{T}[\![p_{n+1}]\!]\,\sigma'' = tt$.

We have $\models \Delta \cup \Delta' \rightarrow \{p_i\}\,A_i\{p_{i+1}\}$; we also have $\mathscr{G}[\![\Delta']\!]\,\gamma = tt$. Now, taking $\gamma_0 = \gamma\{\lambda\sigma\cdot\perp/L_i\}_{i=1}^n$, we can prove that (due to the fact that no L_i occurs in Δ') $\mathscr{G}[\![\Delta']\!]\,\gamma_0 = tt$. Also, $\mathscr{G}[\![\{p_i\}$ **goto**L_i $\{$**false**$\}]\!]\,\gamma_0 = tt$. Thus we have $\mathscr{G}[\![\Delta \cup \Delta']\!]\,\gamma_0 = tt$ and thus $\mathscr{G}[\![\{p_i\}\,A_i\{p_{i+1}\}]\!]\,\gamma_0 = tt$. The same way as in the proof of 8.2a we now get that $\mathscr{T}[\![p_{i+1}]\!]\,\sigma' = tt$ from $\mathscr{A}[\![A_i]\!]\,\sigma = \sigma'$ and $\mathscr{T}[\![p_i]\!]\,\sigma = tt$.

2°. $\mathscr{A}[\![A_i]\!]\,\sigma = \langle\sigma', L_j\rangle$ and thus (4.3.2°) $\sigma'' = \mathscr{N}[\![A_i]\!]\,\gamma^{(k)}\phi_{i+1}^{(k)}\,\sigma = \gamma^{(k)}[\![L_j]\!]\,\sigma' = \phi_j^{(k)}\sigma'$. So, if we can prove $\mathscr{T}[\![p_j]\!]\,\sigma' = tt$, then we can use the induction hypothesis to infer that $\mathscr{T}[\![p_{n+1}]\!]\,\sigma'' = tt$. Wo do this by choosing $\gamma_0 = \gamma\{\bar{\phi}_t/L_t\}_{t=1}^n$, where $\bar{\phi}_t$ is defined by a) $\bar{\phi}_t\sigma = \sigma$, if $\sigma \neq \perp$ and $\mathscr{T}[\![p_t]\!]\,\sigma = ff$; b) $\bar{\phi}_t\sigma = \perp$ otherwise. We again can check that $\mathscr{G}[\![\Delta \cup \Delta']\!]\,\gamma_0 = tt$ and thus $\mathscr{G}[\![\{p_i\}\,A_i\{p_{i+1}\}]\!]\,\gamma_0 = tt$. In a way, analogous to the proof of 8.2b we then can deduce from $\mathscr{T}[\![p_i]\!]\,\sigma = tt$ and $\mathscr{A}[\![A_i]\!]\,\sigma = \langle\sigma', L_j\rangle$ that $\mathscr{T}[\![p_j]\!]\,\sigma' = tt$.

3°. $\mathscr{A}[\![A_i]\!]\,\sigma = \langle\sigma', L\rangle$ where L is not an L_j. Lemma 4.3.2° yields $\sigma'' = \mathscr{N}[\![A_i]\!]\,\gamma^{(k)}\phi_{i+1}^{(k)}\,\sigma' = \gamma^{(k)}[\![L]\!]\,\sigma' = \gamma[\![L]\!]\,\sigma'$ (for $\gamma^{(k)}$ differs from γ only in the arguments $L_1, \ldots, L_n$). Now taking $\gamma_0 = \gamma\{\lambda\sigma\cdot\perp/L_j\}_{j=1}^n$ we have that $\sigma'' = \gamma_0[\![L]\!]\,\sigma'$, but also that $\mathscr{G}[\![\Delta \cup \Delta']\!]\,\gamma_0 = tt$, so that $\mathscr{G}[\![\{p_i\}\,A_i\{p_{i+1}\}]\!]\,\gamma_0 = tt$, or equivalently $\forall\pi \in \Pi \;\; \forall\phi \in M[\{\mathscr{T}[\![p_{i+1}]\!]\}\,\phi\{\pi\} \Rightarrow \{\mathscr{T}[\![p_i]\!]\}\,\mathscr{N}[\![A_i]\!]\,\gamma_0\phi\{\pi\}]$. Now choose $\phi = \lambda\sigma\cdot\perp$ and $\pi = \mathscr{T}[\![p_{n+1}]\!]$. We then have $\{\mathscr{T}[\![p_i]\!]\}(\mathscr{N}[\![A_i]\!]\,\gamma_0\{\lambda\sigma\cdot\perp\})\{\mathscr{T}[\![p_{n+1}]\!]\}$, which combined with $\mathscr{A}[\![A_i]\!]\,\sigma = \langle\sigma', L\rangle$, $\sigma'' = \gamma_0[\![L]\!]\,\sigma'$ and $\mathscr{T}[\![p_i]\!]\,\sigma = tt$ yields $\mathscr{T}[\![p_{n+1}]\!]\,\sigma'' = tt$. □

9. Related Work

First of all the work of Arbib and Alagić [3] should be mentioned. They give a proof system which is quite analogous to the system in Sect. 6. A difference seems to be that their system is structured in such a way that in their correctness formulae $\langle[L_i\colon p_i]_i \,|\, \{p\}\,A\{q\}\rangle$ the invariants corresponding to labels not occurring in A do not have to be specified. However both systems are isomorphic because a formula $\langle L_1\colon p_1 \,|\, \{p\}\,A\{q\}\rangle$, with $L_2, \ldots, L_n$ not in A can be viewed as an abbreviation of $\langle L_1\colon p_1, \ldots, L_n\colon p_n \,|\, \{p\}\,A\{q\}\rangle$ with arbitrary $p_2, \ldots, p_n$. This makes sense: these formulae are either both valid or both invalid due to the fact that $L_2, \ldots, L_n$ are not in A. One can interpret the results of this paper as a justification of Arbib's and Alagić's system as well.

The direct semantics (function $\mathscr{A}$) from Sect. 4 has also been developed by workers which use the Vienna Definition Language ([11], and [7]). Their approach is called **exit** semantics. However they have elaborated this idea to a greater extent. For instance, they define the meaning of a block S as the least fixed point of an operator built up using direct functions, like $\mathscr{A}$. Furthermore they discuss the relative merits and the differences between the two approaches (**exit**s vs. continuations), and they treat several ways to combine the two schemes. In [7] Bjørner suggests that one should choose the style which is most useful for the goals one wants to achieve. Our $\mathscr{A}$-function has been developed precisely with that idea in mind, and therefore this paper forms a nice illustration of Bjørners suggestion.

Then we have the papers by Nassi and Akkoynlu [14], and Kieburtz and Cherniavsky [12], where an axiomatisation of the **break** statement is discussed. Execution of such a statement causes control to be transferred outside the block in which the statement occurs. The cited papers give semantics and soundness and completeness results for these restricted **goto**s.

Finally I like to mention Back's work [4] on the multi-exit statement. Ideas which are related to the ones we discussed in connection with the proof system of Sect. 6 are taken as a starting point for a new programming language which covers a proof system as an intrinsic part.

Acknowledgements. I am grateful to Jaco de Bakker and one of the referees for several remarks that led to improvements. I owe much to Jeff Zucker. He read the manuscript carefully and patiently, and his stimulating remarks had a substantial influence on the final version. Furthermore I like to thank Paul Vitányi for his support in the final stages of this project, and Corrie Klein-Velderman for getting the manuscript in a well-typed form.

References

1. Apt, K.R.: A sound and complete Hoare-like system for a fragment of PASCAL. Report IW 97/78 (preprint), Mathematical Centre, Amsterdam, 1978
2. Apt, K.R., de Bakker, J.W.: Semantics and proof theory of PASCAL procedures. In: Proceedings of the 4th colloquium on automata, languages and programming. Lecture Notes in Computer Science Vol. 52, pp. 30–44, Berlin Heidelberg New York: Springer 1977
3. Arbib, M.A., Alagić, S.: Proof rules for **goto**s. Acta Informat. **11**, 139–148 (1979)
4. Back, R.J.R.: Exception handling with multi-exit statements. In: Programmiersprachen und Programmentwicklungen, (H.-J. Hoffmann, Hrsg.). Informatik-Fachberichte Nr. 25, Berlin Heidelberg New York: Springer 1980
5. Bakker, J.W. de: Mathematical theory of program correctness. London: Prentice Hall 1980
6. Bakker, J.W. de: Correctness proofs for assignment statements. Report IW 55/76, Mathematical Centre, Amsterdam 1976
7. Bjørner, D.: Experiments in block-structured **goto** language modelling: **exit**s versus continuations. In: Winter school on formal specification methods. Lecture Notes in Computer Science Vol. 86 (D. Bjørner, Ed.), Berlin Heidelberg New York: Springer 1980
8. Clint, M., Hoare, C.A.R.: Program proving: jumps and functions. Acta Informat. **1**, 214–224 (1972)
9. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. **7**, 70–90 (1978)
10. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM **12**, 576–580, 583 (1969)
11. Jones, C.B.: Denotational semantics of **goto**: an exit formulation and its relation to continuations. In: The Vienna Development Method: the Meta-Language. Lecture Notes in Computer Science Vol. 61 (D. Bjørner, C.B. Jones, eds.) Berlin Heidelberg New York: Springer

12. Kieburtz, R.B., Cherniavsky, J.C.: Axioms for structural induction on programs containing block exits. In: Proceedings of the Symposium on Computer Software Engineering, Polytechnic Institute of New York, 1976
13. Milne, R., Strachey, C.: A theory of programming language semantics. London: Chapman and Hall, New York: Wiley (2 Vols) 1976
14. Nassi, I.R., Akkoyunlu, E.A.: Semantics of **repeat break** control structure. In: Proceedings of the 8th Princeton Conference on Information Sciences and Systems, pp. 316-321, Princeton University, Dept. of Electrical Engineering New York, 1974
15. Scott, D., Strachey, C.: Towards a mathematical semantics for computer languages. Proc. Symp. on computers and automata, Polytechnic Institute of Brooklyn, pp. 19-46 (1971); also: Techn. Mon. PRG-6, Oxford Univ. Computing Lab.
16. Stoy, J.E.: Denotational semantics - the Scott-Strachey approach to programming language theory. Cambridge, Mass: M.I.T. Press 1977
17. Strachey, C., Wadsworth, C.: Continuations, a mathematical semantics for handling full jumps. Tech. Mon. PRG-11, Oxford Univ. Computing Lab., Programming research group (1974)
18. Wand, M.: A new incompleteness result for Hoare's system. J. Assoc. Comput. Mach. **25**, 168-175 (1978)

Received July 17, 1979/March 31, 1981

chapter 5

THE DENOTATIONAL SEMANTICS OF DYNAMIC NETWORKS OF PROCESSES

The Denotational Semantics of Dynamic Networks of Processes

ARIE DE BRUIN
Erasmus University
AND
WIM BÖHM
University of Utrecht

DNP (dynamic networks of processes) is a variant of the language introduced by Kahn and MacQueen [11, 12]. In the language it is possible to create new processes dynamically. We present a complete, formal denotational semantics for the language, along the lines sketched by Kahn and MacQueen. An informal explanation of the formal semantics is also given.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.3 [Programming Languages]: Language Constructs—*concurrent programming structures, coroutines*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*

General Terms: Languages, Theory

Additional Key Words and Phrases: Continuation semantics, denotational semantics, parallel coroutines, parallellism, recursively defined processes

1. INTRODUCTION

In this paper we define the denotational semantics of DNP (dynamic networks of processes), a language introduced by Kahn and MacQueen [11, 12]. A DNP program describes a network of parallel computing stations (processes) which are interconnected by channels. Processes can only communicate via these channels; there is no sharing of variables. The channels are possibly infinite queues of values. Communication is asynchronous. The computing stations can "expand" into subnetworks, which will be connected to the rest of the network by the original channels. The process that caused the expansion may remain active and become part of the new subnetwork. This is called a *keep*.

Kahn and MacQueen define the meaning of a DNP process as a function from input histories to output histories. A *history* is a possibly infinite sequence modeling the values transmitted through a channel. An intuitive treatment of

Authors' current addresses: A. de Bruin, Faculty of Economics, Erasmus University, P.O. Box 1738, 3000 DR Rotterdam, the Netherlands; W. Böhm, Dept. of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

the semantics of this kind of parallel program is given in [11]. However, it is not specified precisely how to obtain the meaning of a single process from its program text; but an informal treatment is given of how the meaning of a network is derived from the constituent processes and the network topology. In this paper we give a complete formal semantics of the language.

2. SYNTAX

To keep the definition of the semantics short, we use a stripped version of DNP, defined by the following BNF-like syntax.

We use the following syntactic classes as primitives:

$x \in Var$	Program variables
$C \in Chvar$	Channel variables
$P \in Pvar$	Process names
$t \in Exp$	Expressions
$b \in Bexp$	Boolean expressions

Expressions and Boolean expressions are built up from variables, constants, and operators in the usual way. We do not allow function calls inside expressions in order to prevent the evaluation of an expression from having side effects.

$B \in Inst$. *Instantiations*

$$B ::= P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m) \mid \text{keep } P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m).$$

A channel is called an *input channel* if it occurs before the semicolon, and an *output channel* if it occurs after it.

$E \in Ndef$. *Network definitions*

$$E ::= [B_1 \parallel \ldots \parallel B_k],$$

with the restriction that it must be possible to partition the set of all channels occurring in E into three subclasses:

Inchan(E), viz., the channels that occur once and only once as input channels.
Outchan(E), viz., the channels that occur once and only once as output channels.
Intch(E), viz., the internal channels that occur twice, once as input channels and once as output channels.

$S \in Stat$. *Statements*

$$S ::= x := t \mid S_1;\ S_2 \mid \text{while } b \text{ do } S \text{ od} \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{read}(x, C) \mid \text{write}(t, C) \mid \text{expand } E$$

$T \in Decl$. *Process declarations*

$$T ::= P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m) \leftarrow \text{begin } S \text{ end}$$

where

- all C_i are different;
- all channels occurring in a read statement in S are in $\{C_1, \ldots, C_k\}$;

- all channels occurring in a write statement in S are in $\{C_{k+1}, \ldots, C_m\}$;
- for all statements occurring in S of the form expand E:
  Inchan$(E) = \{C_1, \ldots, C_k\}$,
  Outchan$(E) = \{C_{k+1}, \ldots, C_m\}$.
  Every instantiation B in E that is a keep must be of the form keep $P(C'_i, \ldots, C'_k; C'_{k+1}, \ldots, C'_m)$ (i.e., P must be the process name that occurs on the left-hand side of the $\leftarrow$ symbol in the process declaration).

$A \in Prog$. *Programs*

$$A ::= \langle T_1, \ldots, T_n : P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m)\rangle,$$

where $P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m)$, and all instantiations in all T_i are well formed with respect to $T_1, \ldots, T_n$. Here, well-formedness is defined as follows. Let $T_1, \ldots, T_n$ be a sequence of process declarations and (keep) $P'(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m)$ an instantiation B. We call B well-formed with respect to $T_1, \ldots, T_n$ iff there is a T_i in $T_1, \ldots, T_n$ of the form $P'(C'_1, \ldots, C'_k; C'_{k+1}, \ldots, C'_m) \leftarrow$ begin S' end.

Remarks. An expand statement "expand E" occurring in a process declared as

$$T \equiv P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m) \leftarrow \text{begin } S \text{ end}$$

replaces the process T by a subnetwork of processes connected to the rest of the graph by the channels in Inchan(E) ∪ Outchan(E). The processes in the subnetwork are interconnected by the channels in Intch(E), that is, after an expansion all external channels will still be in use and connected to the outside world. The restriction imposed on the class of declarations guarantees these properties for all expand statements. If an instantiation in E is a keep, then the new process inherits the data and control environment of the original process; that is, it will proceed with the statement following "expand E." The other instantiations are fresh copies of processes starting at the first statement with all variables initialised on the value *undefined*. Note that there is no declaration of variables. All variables are strictly local to the process they occur in; there is no sharing.

3. AN EXAMPLE PROGRAM AND ITS ASSOCIATED FUNCTIONS

The following DNP program sorts a sequence of nonnegative numbers followed by −1. This is a simplified version of pipeline sort from [5, Sect. 4.2.1]. Figure 1 depicts the initial network.

A sort process reads one number from the channel "unsorted," creates a fresh sort process in front of it, and inserts the number just read into a sorted subsequence which it expects from the channel "subsequence." The resulting sorted subsequence is written onto channel "sorted." Sort creates a process in front of it by means of the expansion:

```
expand [ sort(unsorted,subsequence1; sorted,empty1)
     ||  keep sort(empty1,subsequence; subsequence1,empty)
     ]
```

which is pictured in Figure 2.

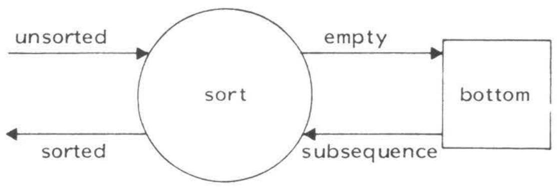

Fig. 1. Initial configuration of the "sort" network.

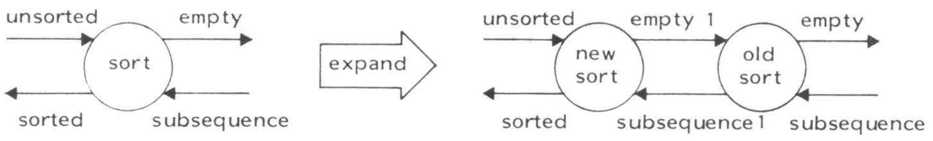

Fig. 2. Expansion of a "sort" process.

The new sort process in Figure 2 is a fresh copy of sort, which will deal with the remaining numbers sent on channel "unsorted." The old sort process is a keep which will manipulate the number it just read. Bottom is a process which just sends an empty (thus sorted) subsequence to the first sort process. Sorting the sequence, 2, 5, 3, −1 proceeds as shown in Figure 3.

We now give the program text.

```
⟨
sort(unsorted,subsequence;sorted, empty) ←
begin read(x,unsorted);
  if x ≥ 0 then expand [ sort(unsorted,subsequence1;sorted,empty1)
                       || keep sort(empty1,subsequence;subsequence1,empty)
                       ] ;
      read(y,subsequence);
      while (y ≥ 0 and y ≤ x)
        do write(y,sorted); read(y,subsequence) od;
      write(x,sorted)
  else read(y,subsequence)
  fi;
  while y ≥ 0 do write(y,sorted); read(y,subsequence) od;
  write(−1,sorted)
end,
bottom(empty;subsequence) ← begin write(−1,subsequence) end,
main(unsorted,sorted) ←
begin expand [ sort(unsorted,subsequence;sorted,empty)
             || bottom(empty;subsequence)
             ]
end
:
main(in;out)
⟩
```

According to [11, 12], we associate the functions from input histories to output histories f_{sort}, f_{bottom}, and f_{main} with the process declarations above. These functions

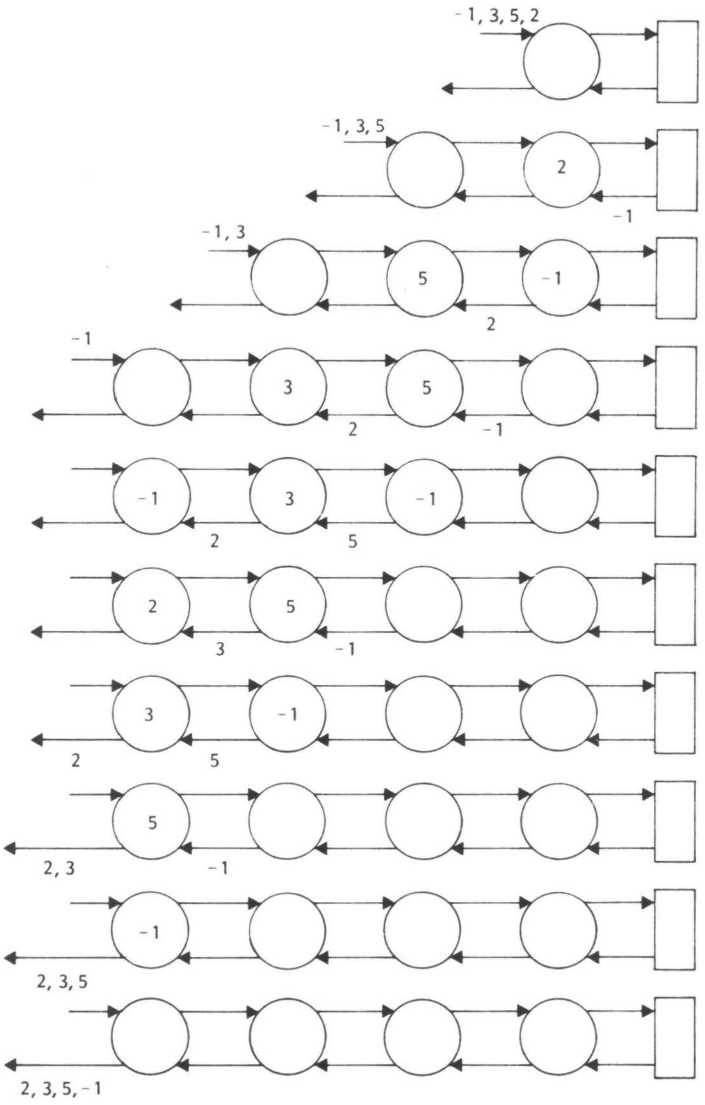

Fig. 3. The "sort" network sorts the sequence 2, 5, 3, −1.

have the following properties ($\hat{}$ denotes concatenation of rows; $.. \rightarrow .., ..$ stands for the if .. then .. else construct).

(1) $f_{\text{bottom}}(X) = \langle -1 \rangle$,

(2) $f_{\text{main}}(X) = Y$,

$$\text{where} \begin{cases} \langle Y, U \rangle = f_{\text{sort}}(X, V) \\ V = f_{\text{bottom}}(U) \end{cases}$$

(3) $f_{\text{sort}}(\langle -1\rangle\hat{\ }X,\langle Y\rangle\hat{\ }Y) = f_{\text{copy}}(y,X,Y)$

(4) $f_{\text{sort}}(\langle x\rangle\hat{\ }X,Y) = \langle U,V\rangle,$

$$\text{where } \left|\begin{array}{l} \langle U,W\rangle = f_{\text{sort}}(X,Z) \\ \langle Z,V\rangle = f_{\text{merge}}(x,W,Y), \quad \text{for} \quad x \geq 0 \end{array}\right.$$

(5) $f_{\text{merge}}(x,X,\langle y\rangle\hat{\ }Y) = (y \geq 0\hat{\ }y \leq x) \rightarrow (\langle y\rangle, \langle\,\rangle)\hat{\ }f_{\text{merge}}(x,X,Y), (\langle x\rangle,\langle\,\rangle)\hat{\ }f_{\text{copy}}(x,X,Y)$

(6) $f_{\text{copy}}(y,X,\langle z\rangle\hat{\ }Y) = (y \geq 0) \rightarrow (\langle y\rangle,\langle\,\rangle)\hat{\ }f_{\text{copy}}(z,X,Y), (\langle -1\rangle\ \langle\,\rangle)$

We now present an informal justification of these equations. The first one is easy: the process bottom does not read from its input channel, and the only item it writes on its output channel is the value -1. So f_{bottom} is a constant function that yields for all input histories X the one element history $\langle -1\rangle$. The behavior of main is also straightforward. This process expands into a network consisting of a sort and a bottom process. For the output history Y of f_{main}, (2) must hold. Here X corresponds to the channel "unsorted," Y to "sorted," U to "empty," and V to "subsequence." The complicated case is the behavior of sort, which can be described in three stages. During the last stage, corresponding to the last three lines in the declaration of sort, the input from "subsequence" is copied to the output channel "sorted." This behavior is captured in the function f_{copy} in (6). The first argument of this function corresponds to the value of the variable y, the last value read form "subsequence." Now (3) follows as the only effect of the process sort when it reads -1 from "unsorted" is to read the first element from "subsequence" and do the copying.

The last eight lines in the declaration of sort describe a merge process captured by the function f_{merge}. First, all values from "subsequence" not greater than x (the value read from input channel "unsorted") are copied to "sorted." Then x is inserted in "sorted," and finally the rest of the "subsequence" is copied to "sorted," as described by f_{copy}. All this is captured in (5). Finally, the expansion in sort is described by (4). A fresh copy of sort is generated, described by f_{sort}, and the keep process will execute the body of sort after the expand statement. This last behavior is captured by f_{merge}. In (4), x corresponds to the value of the local variable x in sort, which is equal to the value just read from "unsorted," and X, Y, U, V, W, and Z correspond to the channels "unsorted," "subsequence," "sorted," "empty," "empty1," and "subsequence1" respectively. The meaning of the program is the meaning of the initial network, viz., f_{main}.

In these equations three forms of recursion occur. The simplest is the recursive definition of f_{merge} in (5), which stems from the fact that f_{merge} models the behavior of a while statement. In (4), two kinds of recursion can be observed. First, the histories Z and W are defined recursively because the subnetwork in which they occur is cyclic. Second, f_{sort} is defined recursively, which stems from the fact that f_{sort} models the behavior of an expand statement in sort. In [11], Kahn showed informally that the behavior of the system will in fact be captured by the smallest solution of the above equations (i.e., the solution with the shortest output histories).

In the formal semantics in Section 4 these recursive definitions are taken care of by the least fixed-point definitions 4.2.2.4, 4.2.2.7, and 4.2.6, respectively.

The above equations are sufficient to show that the network indeed yields a sorted permutation of the input sequence [5, Sect. 5.2]. The fact that the system corresponds to the smallest solution of (1)–(6) is not needed for the proof, which runs as follows. We have to prove that $f_{\mathrm{sort}}(X, Y)$ yields a sorted permutation of X and Y, provided that Y is sorted and that X and Y are well formed (i.e., consist of positive integers and a final -1). This can be done using (4) by induction on the length of X. We need a proposition on the behavior of f_{merge} for this proof, stating that if Y is sorted and well formed then $f_{\mathrm{merge}}(x, X, Y)$ yields a sorted permutation of $\langle x \rangle \hat{} Y$. This proposition can be proved by induction on the length of Y, using Eq. (5) and a similar proposition on f_{copy}.

4. SEMANTICS

In this section we present concisely the semantical domains and functions. The next section is devoted to some explanatory remarks. The reader is invited to skim the definitions first, and afterwards study them in the order suggested by the explanations in Section 5.

We make use of the following notational conventions:

- If X and Y are sets, then $X \rightarrow Y$ denotes the set of all functions from X to Y. If, moreover, X and Y are cpos (complete partial orders), then $[X \rightarrow Y]$ denotes the cpo of all continuous functions in the cpo $X \rightarrow Y$.
- Function applications associate to the left (e.g., $fabc = ((f(a))(b))(c))$.
- The $\rightarrow$ operator associates to the right (e.g., $A \rightarrow B \rightarrow C \rightarrow D = A \rightarrow (B \rightarrow (C \rightarrow D)))$.
- To enhance readability, syntactical arguments are enclosed in $[\![\,]\!]$-type brackets and continuations in $\{\}$-type brackets.
- $\alpha \rightarrow \beta, \gamma$ denotes β if α is true, and γ otherwise.
- If $f \in X \rightarrow Y$, then $f[y/x]$ denotes the function $\lambda x'.(x' = x) \rightarrow y, fx'$. We also employ a parallel version: $f[gx/x]_{x \in X} = \lambda x'.(x' \in X) \rightarrow gx', fx'$.
- If V is a set, then V^{∞} denotes the cpo of all finite and infinite sequences of values from V, ordered by the relation "is a prefix of."
- Tuple notation: the sequence of objects $x_1, \ldots, x_n$ is denoted by $\langle x_1, \ldots, x_n \rangle$. Concatenation is denoted by $\hat{}$. Projection is denoted by subscripts or by a down arrow $\downarrow$ (e.g., if $x = \langle a, b, c \rangle$, then $x_2 = b$ and $x \downarrow 3 = c$).
- μ denotes the least fixed-point operator (e.g., if X is a cpo and $f \in X \rightarrow X$ is continuous, then μf is the smallest element in X such that $f(\mu f) = \mu f$).

4.1 Domains

Values	$\delta \in V \quad (\mathit{undefined} \in V)$
States	$\sigma \in \Sigma = \mathit{Var} \rightarrow V$
Histories	$\tau \in V^{\infty}$
Channel contents	$\epsilon \in \mathit{Chcont} = \mathit{Chvar} \rightarrow V^{\infty}$
Processes	$\alpha \in \mathit{Process} = [\mathit{Chcont} \rightarrow \mathit{Chcont}]$
Continuations	$\theta \in \mathit{Cont} = \Sigma \rightarrow \mathit{Process} = \Sigma \rightarrow [\mathit{Chcont} \rightarrow \mathit{Chcont}]$
Process generators	$\beta \in \mathit{Prgen} = \mathit{Chvar}^* \rightarrow \mathit{Chvar}^* \rightarrow \mathit{Process}$
Environments	$\gamma \in \mathit{Env} = (\mathit{Pvar} \rightarrow \mathit{Prgen}) \times (\mathit{Pvar} \rightarrow [\mathit{Process} \rightarrow \mathit{Prgen}])$

4.2 Functions

4.2.1 $M: Exp \rightarrow \Sigma \rightarrow V$ and $M: Bexp \rightarrow \Sigma \rightarrow \{\text{true, false}\}$ are assumed to be predefined.

4.2.2 $M: Stat \rightarrow [Env \rightarrow [Cont \rightarrow (\Sigma \rightarrow Process)]]$

4.2.2.1 $M[\![x := t]\!]\gamma\theta\sigma\epsilon = \theta(\sigma[M[\![t]\!]\sigma/x])\epsilon$

4.2.2.2 $M[\![S_1; S_2]\!]\gamma\theta\sigma\epsilon = M[\![S_1]\!]\gamma\{M[\![S_2]\!]\gamma\theta\}\sigma\epsilon$

4.2.2.3 $M[\![\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}]\!]\gamma\theta\sigma\epsilon = M[\![b]\!]\sigma \rightarrow M[\![S_1]\!]\gamma\theta\sigma\epsilon, M[\![S_2]\!]\gamma\theta\sigma\epsilon$

4.2.2.4 $M[\![\text{while } b \text{ do } S \text{ od}]\!]\gamma\theta\sigma\epsilon = M[\![b]\!]\sigma$
$\rightarrow M[\![S]\!]\gamma\{M[\![\text{while } b \text{ do } S \text{ od}]\!]\gamma\theta\}\sigma\epsilon, \theta\sigma\epsilon$

4.2.2.5 $M[\![\text{read}(x, C)]\!]\gamma\theta\sigma\epsilon = (\epsilon C = \langle\,\rangle) \rightarrow \lambda C.\langle\,\rangle, \theta\sigma'\epsilon'$,
where $\sigma' = \sigma[\mathit{first}(\epsilon C)/x]$ and $\epsilon' = \epsilon[\mathit{rest}(\epsilon C)/C]$

4.2.2.6 $M[\![\text{write}(t, C)]\!]\gamma\theta\sigma\epsilon = \lambda C'. (C' \equiv C) \rightarrow \langle M[\![t]\!]\sigma\rangle\,\hat{}\,\theta\sigma\epsilon C', \theta\sigma\epsilon C'$

4.2.2.7 $M[\![\text{expand } E]\!]\gamma\theta\sigma\epsilon = \lambda C.(C \in \text{Outchan}(E)) \rightarrow (\mu\Phi^*)C, \langle\,\rangle$,
where $\Phi^*: Chcont \rightarrow Chcont$ is defined by
$\Phi^*\epsilon' = M[\![E]\!]\gamma\theta\sigma(\epsilon'[\epsilon C/C]_{C\epsilon\text{Inchan}(E)})$

4.2.3 $M: Inst \rightarrow [Env \rightarrow [Cont \rightarrow (\Sigma \rightarrow Process)]]$

4.2.3.1 $M[\![P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m)]\!]\gamma\theta\sigma = \gamma_1 P\langle C_1, \ldots, C_k\rangle\langle C_{k+1}, \ldots, C_m\rangle$

4.2.3.2 $M[\![\text{keep } P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m]\!]\gamma\theta\sigma$
$= \gamma_2 P(\theta\sigma)\langle C_1, \ldots, C_k\rangle\langle C_{k+1}, \ldots, C_m\rangle$

4.2.4 $M: Ndef \rightarrow [Env \rightarrow [Cont \rightarrow (\Sigma \rightarrow Process)]]$
$M[\![[B_1 \parallel \cdots \parallel B_k]]\!]\gamma\theta\sigma = concat(M[\![B_1]\!]\gamma\theta\sigma, \ldots, M[\![B_k]\!]\gamma\theta\sigma)$,

$$\text{where } concat(\alpha_1, \ldots, \alpha_k)\epsilon C = \begin{cases} \alpha_i\epsilon C \text{ for the smallest } i \text{ such that} \\ \qquad \alpha_i\epsilon C \neq \langle\,\rangle, \text{ if there is such an } i \\ \langle\,\rangle \quad \text{otherwise} \end{cases}$$

4.2.5 $M: Decl \rightarrow [Env \rightarrow Env]$
$M[\![P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m) \leftarrow \text{begin } S \text{ end}]\!]\gamma$
$= \langle\gamma_1[\phi_1/P], \gamma_2[\phi_2/P]\rangle$,
where $\phi_2 = \lambda\alpha.\lambda\langle C_1', \ldots, C_s'\rangle.\lambda\langle C_{s+1}', \ldots, C_t'\rangle.$
$(s \neq k \text{ or } t \neq m) \rightarrow \lambda\epsilon.\lambda C.\langle\,\rangle,$
$\lambda\epsilon.\lambda C.(C \equiv C_{s+i}') \rightarrow \alpha\epsilon' C_{s+i}, \langle\,\rangle$
where $\epsilon' = \lambda C.(C \equiv C_i) \rightarrow \epsilon C_i', \langle\,\rangle$
and $\phi_1 = \phi_2(M[\![S]\!]\gamma\{\lambda\sigma.\lambda\epsilon.\lambda C.\langle\,\rangle\}(\lambda x.\mathit{undefined}))$

4.2.6 $M: Prog \rightarrow Process$
$M[\![T_1, \ldots, T_n: P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m))]\!]$
$= (\bar{\gamma} \downarrow 1)P\langle C_1, \ldots, C_k\rangle\langle C_{k+1}, \ldots, C_m\rangle$,
where $\bar{\gamma} = (M[\![T_1]\!]^\circ \ldots {}^\circ M[\![T_n]\!])\bar{\gamma}$

. . . I said: "Well, what about the other people in the world who might enjoy the melody of the black page but could not really approach its statistical density in its basic form." So I went to work . . .

Frank Zappa
The Black Page, #2

5. DISCUSSION

In the headings of the following subsections we refer to the corresponding semantic clauses from Section 4. We assume acquaintance with the concepts of denational semantics as provided in, for example, [10, 13, 17].

5.1 Domains (4.1)

5.1.1 *Values, States and Histories.* V denotes the set of all values that can be assumed by the program variables. One special value *undefined* is added because we do not want to be bothered by nonessential nondeterminism caused by uninitialized variables; we initialize all variables with *undefined*. Note that V is not a cpo and, in particular, that *undefined* does not act as the bottom element. In fact, since we did not define the function $M: Exp \rightarrow \Sigma \rightarrow V$ and $M: Bexp \rightarrow \Sigma \rightarrow \{\text{true, false}\}$, we left unspecified what will happen when one the variables in an expression is undefined. (What if we test $x = y$ and both x and y are not initialized?) We do not work this out here because the emphasis is on the semantics of concurrency and process creation. For the same reason we do not make V a cpo. We could have included the value $\perp$ modeling a nonterminating evaluation of an expression. Some remarks on the consequence of introducing these features are made in Section 7. States are defined in the usual way. Each process has its own state; there is no sharing of variables between processes. Note that Σ is a set and not a cpo. A history is a finite or infinite sequence of values. On the class of histories we do impose a cpo structure by defining $\tau_1 \sqsubseteq \tau_2$ iff τ_1 is a prefix of τ_2. The bottom element in V^∞ is the empty sequence $\langle\ \rangle$, and all infinite sequences are maximal (if $\tau_1 \sqsubseteq \tau_2$ and τ_1 is infinite, then $\tau_1 = \tau_2$). In contrast to the approach in sequential programs, a nonterminating process is not modeled by a bottom element $\perp$, but by one or more infinite output histories (provided of course that the nonterminating process does not stop generating output).

5.1.2 *Channel Contents and Processes.* In Section 3 we defined (like Kahn and MacQueen) the meaning of a process as a function from tuples of histories to tuples of histories. Our approach follows these lines, but as we define the semantics of a process declaration in a compositional way (by induction on the structure of its body), we cannot easily use functions on tuples of histories because when we define the meaning of a statement containing a channel variable, the position of that channel variable in the input or output tuple is no longer known. Instead, we apply the mechanism as customary for states: the meaning of a statement is a function from channel contents to channel contents, where a channel contents associates a history with every channel variable.

So the meaning of a program A will be a process, a function that might be called a "channel contents transformer." Such a process α takes a channel contents $\epsilon \in$ *Chcont*, which models the histories on its inputs channels and yields the resulting histories on its output channels, consisting of all values written there by the program. It has been justified in [11] that denotations of programs are continuous, so we allow only continuous functions from *Chcont* to *Chcont* in *Process*. Notice that there are infinite objects in *Chcont*, the results of infinite computations. Usually infinite computations are modeled by a bottom element, but our semantics yields a well-defined and useful result: we are not interested in a "final" result, but rather in the sequence of outputs produced during the computation.

5.1.3 *Process Generators and Environments.* In the end, the meaning $M[\![S]\!]$ of a statement will be a process, but this process depends on the meaning of the process names occurring in S. We therefore have to provide $M[\![S]\!]$ with an argument, namely, an environment. A process declaration yields a "formal process" which is a process in terms of the formal channel names, but an instantiation must yield a process in terms of the actuals. To this end, the domain of process generators is introduced. A generator accepts a finite list of actual input channels and a finite list of actual output channels and yields the actual process.

For a normal instantiation (i.e., not a keep) the formal process, and thus the corresponding process generator, is derived from its declaration. This is modeled by the first component of an environment. For a keep the formal process must be supplied explicitly (because it is defined by the execution of the rest of the program, following the expand statement which contains the keep), and this is modeled by the second component of an environment.

5.1.4 *Continuations.* Direct semantics does not seem appropriate. Consider the meaning of composition. This should be something like

$$M[\![S_1;\, S_2]\!]\gamma\sigma\epsilon = M[\![S_2]\!]\gamma(M[\![S_1]\!]\gamma\sigma\epsilon) = \epsilon',$$

where σ is the initial state, ϵ models the contents on the input channels, and ϵ' models the result on the output channels. Now $M[\![S_1]\!]\gamma\sigma\epsilon$ must yield an intermediate result, and this poses at least three problems:

(i) What if S_1 blocks on trying to read from an empty channel? A special intermediate state *blocked* could be introduced, but this can hardly be called an elegant solution.

(ii) An intermediate result must at least contain an intermediate state σ', an intermediate contents of the input channels, and the output resulting from S_1. Now $M[\![S_2]\!]$ must concatenate its own output to S_1's output, but concatenation $(\lambda\tau_1.\lambda\tau_2.\tau_1\hat{\ }\tau_2)$ is not continuous.

(iii) What should $M[\![\text{expand } E]\!]$ look like?

We therefore use continuation semantics. We give $M[\![S]\!]\gamma$ an extra argument θ, a continuation, which is meant to model the future of the computation (i.e., θ supplies the meaning of the statements to be executed after S). In other words, if θ specifies how execution proceeds once the right-hand end of S has been reached, $M[\![S]\!]\gamma\theta$ specifies execution starting from the left-hand end of S. Continuations are due to Strachey and Wadsworth [18] and Morris [14]. An easy introduction to this topic can be found in [10]. More technical information appears in [13, 17]. The domain of all continuations is *Cont*: the future of a computation is modeled by a θ, which takes a state and a contents of input channels and yields the contents of the output channels.

5.2 The Function M

5.2.1 *Instantiations* $M[\![B]\!]$ (4.2.3). An instantiation is always part of an expand statement. An instantiation either creates a fresh copy of a process (normal instantiation) or resumes the process in which the expand statement occurs (keep). The meaning $M[\![B]\!]\gamma\theta\sigma$ of an instantiation B in an environment γ is a process $\alpha \in$ *Process* which corresponds to executing the body of B (see also Section 5.1.3). The arguments θ and σ are those associated with the expand statement in which the instantiation occurs. For normal instantiations we obtain the process generator from the first component of the environment and the process name. We then apply this generator to the actual channels, and this yields the actual process.

A keep corresponds to the expanding process, which remains active after execution of the expansion. This process will start executing the statements (dynamically) following the expand statement, and is therefore described by the continuation θ associated with the expand statement. The starting state is the state σ in which the original process expanded. So the formal process we need is $\theta\sigma$.

5.2.2 *Declarations* $M[\![T]\!]$ (4.2.5). The meaning $M[\![T]\!]\gamma$ of a declaration T in an environment γ is a new environment: with a process name P two functions ϕ_1 and ϕ_2 are associated (see also the discussion of the domain *Env* in Section 5.1.3). First, ϕ_2; this function expects a process α specified in terms of the formal input and output channels C_i, and two lists of actual channel names C_i'. It yields the actual process. This formal-actual transformation proceeds in two stages. The contents of the actual input channels are given by ϵ. First ϵ is transformed to ϵ', which models the same input, but now in terms of the formals. Thus $\alpha\epsilon'$ yields the right output, but in terms of the formals. This is rewritten to an element of *Chcont* in terms of the actuals in the λ-expression:

$$\lambda\epsilon.\lambda C.(C \equiv C'_{s+i}) \longrightarrow \alpha\epsilon' C_{s+i}, \langle\ \rangle.$$

For the function ϕ_1, a formal process does not have to be supplied explicitly. It will be derived from the declaration T by evaluating its body with respect to the empty continuation (after execution of the body no further writes will occur on the output channels) in an initial state where all variables are undefined. Notice that all process generators β, which can be defined through Section 4.2.5, will write on their output channels only; that is, for all ϵ we have $\beta\langle C_1, \ldots, C_n\rangle * \langle C_1', \ldots, C_k'\rangle\epsilon C = \langle\ \rangle$ unless $C \equiv C_i'$.

5.2.3 *Programs* $M[\![A]\!]$ (4.2.6). The meaning of a program is the meaning of its body evaluated in the environment determined by the declarations. Notice that the definition is recursive in $\bar{\gamma}$. This is needed because there can be recursive instantiations in the bodies of the T_i.

5.2.4 *Statements* $M[\![S]\!]$ (4.2.2). $M[\![S]\!]\gamma\theta\sigma\epsilon$ yields a channel contents ϵ' describing the histories on the output channels resulting from executing S followed by the future computation as described by the continuation θ. S is executed in a state σ with respect to an environment γ where the contents of the input channels are given by ϵ.

5.2.4.1 *Assignment* (4.2.2.1). $M[\![x := t]\!]\gamma\theta\sigma\epsilon$ yields the contents of the output channels by first of all evaluating the assignment $x := t$ in σ (yielding an updated state $\sigma[M[\![t]\!]\sigma/\mathrm{x}]$), and after that proceeding as given by the continuation θ.

Therefore, the effect of an assignment is captured by applying the continuation to the updated state. The contents of the input channels do not change because no input is read.

5.2.4.2 *Composition* (4.2.2.2). Composition is handled in the standard way: evaluation of S_1; S_2 with respect to θ is equivalent to evaluation of S_1 with respect to {evaluation of S_2 with respect to θ}, cf. 5.1.4.

5.2.4.3 *Conditional* (4.2.2.3). The result of evaluating "if b then S_1 else S_2 fi" with respect to environment γ, continuation θ, state σ, and input channel ϵ is either the result of evaluating S_1 with respect to these parameters (namely, if b evaluated in σ yields true) or the results of evaluating S_2 (otherwise).

5.2.4.4 *Repetition* (4.2.2.4). The statement "while b do S od" is equivalent to

```
"if b then S; while b do S od
    else skip
    fi"
```

Evaluating the meaning of the latter statement gives us 4.2.2.4. Notice the recursion here. Equation 4.2.2.4 is an informal way of writing down the least fixed-point expression

$$M[\![\text{while } b \text{ do } S \text{ od}]\!]\gamma\theta = \mu[\lambda\theta'.\lambda\sigma.M[\![b]\!]\sigma \rightarrow M[\![S]\!]\gamma\theta'\sigma,\, \theta\sigma].$$

A similar remark applies to the definition of $\bar{\gamma}$ in the definition of the meaning of programs (4.2.6).

5.2.4.5 *Input* (4.2.2.5). In defining the meaning of a read statement two cases can be discriminated. If the input channel is empty, the process is blocked; it will have no effect on its output channels (i.e., it yields $\lambda C.\langle\ \rangle$). As the process is blocked, the continuation, which models the future of the computation, must be ignored. Remember that our semantics assumes that all input that will be supplied to a process is given by the initial channel contents, there is no such thing modeled in our semantics as a process waiting for input.

If the input channel is not empty, then read (x, C) is equivalent to the assignments

$$x := \text{first element of } C; \qquad C := \text{rest of } C.$$

5.2.4.6 *Output* (4.2.2.6). Consider the write statement "write(t, C)" evaluated with respect to a continuation θ. For all channels except C this statement is equivalent to the empty statement. The output history on C consists of the value of t followed by what will be written on C in the future.

A discussion of the expand statement will be given after we have treated network definitions.

5.2.5 *Network definitions* $M[\![E]\!]$ (4.2.4). To model the expand statement we need to find the (smallest) solution of a set of equations in history-valued

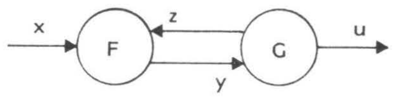

Fig. 4. A typical network.

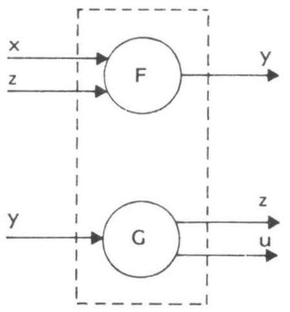

Fig. 5. The network of Figure 4 unfolded.

variables, derived from the topology of the new network. Consider as an example an expansion into the net in Figure 4.

According to Kahn, the global behavior of the net is described by an operator which takes an input history x and yields an output history u. This operator is derived by solving the equations

$$y = F(x, z)$$
$$\langle z, u \rangle = G(y)$$

This is equivalent to deriving the least fixed-point of $\lambda \langle y, z \rangle . \langle F(x, z), G(y) \downarrow 1 \rangle$, where $G(y) \downarrow 1$ corresponds to output on the channel labelled z. In our approach we follow the same line of thought, but now in terms of channel contents. This means that we need to find the least fixed-point of an operator from *Chcont* to *Chcont*. This is accomplished in two stages. First we describe the behavior of the processes in the network as if they were not interconnected (i.e., the internal channels occur twice but the two occurrences are not yet related). In terms of the example above we derive the operator $M[\![E]\!]$, which is pictured in Figure 5. This is what Eq. 4.2.4 describes.

Let us try to show this result more precisely. The network in Figure 4 can be expressed in our syntax as the statement

$$\text{expand } E,$$

where

$$E \equiv [\text{procF}(X, Z;\ Y) \parallel \text{procG}(Y;\ Z, U)].$$

Notice that Inchan(E) = $\{X\}$. The processes procF and procG must be declared such that it will yield an environment γ with

$$\gamma_1[\text{procF}] = \beta_F \quad \text{and} \quad \gamma_1[\text{procG}] = \beta_G.$$

The process generators β_F and β_G should yield processes that correspond to the functions F and G, but stated in terms of elements from *Chcont*. This means that

$$\beta_F\langle X, Z\rangle\langle Y\rangle = \alpha_F$$
$$\beta_G\langle Y\rangle\langle Z, U\rangle = \alpha_G$$

where

$$\alpha_F\epsilon Y = F(\epsilon X, \epsilon Z) \quad \text{and} \quad \alpha_F\epsilon C = \langle\,\rangle \quad \text{for} \quad C \neq Y$$
$$\langle\alpha_G\epsilon Z, \alpha_G\epsilon U\rangle = G(\epsilon Y) \quad \text{and} \quad \alpha_G\epsilon G = \langle\,\rangle \quad \text{for} \quad C \neq Z, U.$$

According to [11], the network in Figure 4 corresponds to a function f_{net}, where $f_{\text{net}}(x) = u = G(y) \downarrow 2$, with $\langle y, z\rangle$ the smallest solution of

$$y = F(x, z)$$
$$z = G(y) \downarrow 2,$$

or equivalently $f_{\text{net}}(x) = u$, where u is the last component of the least fixed-point of the operator 0, defined by

$$0(y, z, u) = \langle F(x, z), G(y) \downarrow 1, G(y) \downarrow 2\rangle$$

(notice that 0 depends on x).

$M[\![\text{expand } E]\!]$ should thus yield the same result, or, more precisely, it must be the process $\alpha_{\text{net}} \in$ *Process* corresponding to f_{net} in the same way that α_F corresponds to F.

We first consider $M[\![E]\!]\gamma\theta\sigma$, with γ equipped with the right values in procF and procG, and θ and σ arbitrary. According to 4.2.4,

$$M[\![E]\!]\gamma\theta\sigma = concat(M[\![\text{procF}(X, Z;\, Y)]\!]\gamma\theta\sigma, M[\![\text{procG}(Y;\, Z, U)]\!]\gamma\theta\sigma).$$

Using 4.2.3.1 and the above definitions of β_F, β_G, α_F, and α_G, we get

$$\begin{aligned} M[\![E]\!]\gamma\theta\sigma &= concat(\beta_F\langle X, Z\rangle\langle Y\rangle, \beta_G\langle Y\rangle\langle Z, U\rangle) \\ &= concat(\alpha_F, \alpha_G) = \lambda\epsilon.\epsilon', \end{aligned}$$

where

$$\begin{aligned} \epsilon' Y &= F(\epsilon X, \epsilon Z) \\ \langle\epsilon' Z, \epsilon' U\rangle &= G(\epsilon Y) \\ \epsilon' C &= \langle\,\rangle, \quad \text{for all other } C\text{s}. \end{aligned}$$

This operator corresponds to the one sketched in Figure 5. We call this operator α_E. The next stage consists of "connecting the internal channels" in α_E; and that is what $M[\![\text{expand } E]\!]$ is supposed to realize.

5.2.6 *Expand Statement* $M[\![\text{expand } E]\!]$ (4.2.2.7). Here is the second stage. We have derived an operator $M[\![E]\!]\gamma\theta\sigma$ and now we will transform it into an operator Φ^*, of which we will take the fixed point. Φ^* essentially connects the internal channels. In terms of the example, Φ^* rephrases the operator $\lambda\langle y, z\rangle.\ \langle F(x, z), G(y) \downarrow 1\rangle$ as a function from *Chcont* to *Chcont*. Note that we cannot simply take the fixed point of $M[\![E]\!]\gamma\theta\sigma$ because the global input given by ϵ in the definition must be supplied explicitly. Note also that the results on the internal channels are invisible from outside the expand statement.

We now continue our derivation, started at the end of Section 5.2.5, and show that $M[\![\text{expand } E]\!]$ indeed yields the desired process α_{net}. According to 4.2.2.7,

$$M[\![\text{expand } E]\!]\gamma\theta\sigma = \alpha_n,$$

with

$$\alpha_n\epsilon = \lambda C.(C \in \text{Outchan}(E)) \longrightarrow (\mu\Phi^*)C, \langle\ \rangle$$

or, since $\text{Outchan}(E) = \{U\}$,

$$\alpha_n\epsilon = \lambda C.(C \equiv U) \longrightarrow (\mu\Phi^*)U, \langle\ \rangle.$$

So we must show that the operator Φ^* corresponds to the operator 0 as defined in 5.2.5. Notice that Φ^* depends on ϵ just as 0 depends on x. We therefore choose an $\epsilon \in \mathit{Chcont}$ such that $\epsilon X = x$, the history used in the definition of 0. According to 4.2.2.7, again we have

$$\Phi^*\epsilon'' = M[\![E]\!]\gamma\theta\sigma(\epsilon''[\epsilon C/C]_{C\in\text{Inchan}(E)}).$$

As $\text{Inchan}(E) = \{X\}$ and $M[\![E]\!]\gamma\theta\sigma = \alpha_E$, we get

$$\Phi^*\epsilon'' = \alpha_E(\epsilon''[\epsilon X/X]),$$

or, using the results on α_E,

$$\Phi^*\epsilon''Y = F((\epsilon''[\epsilon X/X])X, (\epsilon''[\epsilon X/X])Z) = F(\epsilon X, \epsilon''Z)$$
$$\langle \Phi^*\epsilon''Z, \Phi^*\epsilon''U\rangle = G(\epsilon''Y).$$

In other words, Φ^* is an operator that takes a channel contents ϵ'' and yields an $\epsilon'' \in \mathit{Chcont}$ which is empty on all channels except Y, Z, and U, where it assumes the above values.

This means that Φ^* is indeed equivalent to 0, and therefore the least fixed-point $\mu\Phi^*$ yields the same values in Y, Z, and U as $\mu 0$. Or, more precisely, we have

$$\langle(\mu\Phi^*)Y, (\mu\Phi^*)Z, (\mu\Phi^*)U\rangle = \mu 0.$$

5.2.7 *The Existence of M.* We have to show that M is well defined in the sense that for any syntactical object Δ, $M[\![\Delta]\!]$ is an element of the right domain (e.g., for every instantiation B we have it that $M[\![B]\!]\gamma\theta\sigma\epsilon$ must be continuous in γ, θ, and ϵ). This result then guarantees the existence of the fixed points occurring in the definition.

These properties can be shown by induction on the complexity of Δ, but there are some subtleties to deal with. They stem from the fact that we have to be careful in dealing with conditional clauses $\alpha \longrightarrow \beta, \gamma$ because they are not continuous in their first argument. So we have to check these occurrences in Section 4. No danger exists in 4.2.2.6, 4.2.2.7, and 4.2.5 (write, expand, and declarations) because the tests contain syntactical entities only. Neither do we have to deal with continuity in 4.2.2.3 and 4.2.2.4 (if and while) because we did not make V, and therefore Σ, a cpo. This was a deliberate choice, which is discussed in Section 6. As regards 4.2.2.5 (read), we have to be careful, because the test depends on ϵ. However, if we turn {true, false} into a cpo by defining true $\sqsubseteq$ false, then everything works.

More care has to be exercised in dealing with 4.2.4 (network definitions), because as it stands $concat(\alpha_1, \ldots, \alpha_k)\epsilon$ is not continuous in either α_i or ϵ. In fact, *concat* is not even monotonic, which can be seen by taking $\epsilon_1 C = \langle\,\rangle$, $\epsilon_2 C = \langle x \rangle$, $\alpha_1 = \lambda\epsilon.\epsilon$ and $\alpha_2 = \lambda\epsilon.\lambda C.\langle y \rangle$, where $x \neq y$. Now $concat(\alpha_1, \alpha_2)\epsilon_1 C = \langle y \rangle$ and $concat(\alpha_1, \alpha_2)\epsilon_2 C = \langle x \rangle$. The problem here is that α_i and α_j might write on the same output channel. In fact, this is the only obstacle; if we assume that the α_i are such that an output clash cannot occur, then *concat* is continuous. Also, the discontinuity of *concat* does not seem to be essential in the sense that, after a finite number of approximations ϵ_i, $concat(\alpha_1, \ldots, \alpha_k)\epsilon_i$ will "choose" the right and final α_j (i.e., the same as $concat(\alpha_1, \ldots, \alpha_k)(\sqcup\epsilon_i)$). In fact, we can make *concat* continuous by providing V^∞ with a richer structure than that of cpo, (see also [6, Sect. 6], where an idea like this has been worked out). Here we solve the difficulty in another way, by realizing that the possible arguments $M[\![B_i]\!]\gamma\theta\sigma$ of *concat* in 4.2.4 can only write on the output channels occurring in B_i (cf. the remark at the end of Section 5.2.2). Now every network definition $[B_1 \parallel \cdots \parallel B_k]$ must be well formed (cf. the definition of *Ndef* in Section 2), and this means that the sets of output channels of the respective B_is will be disjoint.

In order to prove that $M[\![\Delta]\!]$ is in the right domain, we have to adopt this observation in the induction hypothesis. This can be worked out as follows.

Definition 1. A process generator β *writes on its output channels only* iff for all tuples $\langle C_1, \ldots, C_k \rangle$, $\langle C_1', \ldots, C_m' \rangle$ and for all ϵ and C we have that

$$\beta\langle C_1, \ldots, C_k\rangle\langle C_1', \ldots, C_m'\rangle\epsilon C = \langle\,\rangle \text{ unless } C \equiv C_i' \text{ for some } i.$$

Definition 2. An environment γ is *clean* iff for all P and for all α we have that $\gamma_1 P$ and $\gamma_2 P\alpha$ write on their output channels only.

PROPOSITION. *Let Δ be a syntactic entity in Stat ∪ Inst ∪ Ndef ∪ Decl. If we restrict Env to the domain of all clean environments, then*

(1) *$M[\![\Delta]\!]$ is in the right domain;*
(2) *if $\Delta \equiv B \in$ Inst, then $M[\![B]\!]\gamma\theta\sigma\epsilon$ writes only on the output channels occurring in B.*

PROOF. Induction on the structure of Δ. Notice that we have to check that *Env* restricted as above is a cpo (i.e., the hub of a chain of clean environments must itself be clean). However, this follows immediately from a similar fact on process generators. Notice also that in case $\Delta \equiv T$, (1) denotes that $M[\![T]\!]\gamma$ is clean if γ is clean. □

THEOREM. *For all programs A, its meaning $M[\![A]\!]$ is well defined.*

PROOF. Let $A \equiv \langle T_1, \ldots, T_n : P(C_1, \ldots, C_k; C_{k+1}, \ldots, C_m)\rangle$. We have to prove (according to 4.2.6) that the least fixed-point $\bar{\gamma} = \mu\Gamma$ exists, where $\Gamma = \lambda\gamma.\,(M[\![T_1]\!] \circ \cdots \circ M[\![T_n]\!]\gamma)$. This is established as follows.

(1) Γ transforms a clean environment into a clean environment (proposition (1), for $\Delta \equiv T$).
(2) Γ is a continuous operator from clean environments to clean environments (again proposition (1); note that the bottom element of *Env* is clean as

well). In fact this analysis shows that all environments that correspond to valid declarations $T_1, \ldots, T_n$ are clean; that is, all environments that can occur in the inductive definition of a program using the clauses from Section 4 are clean.

6. AN APPLICATION

The semantics we have constructed are not very well suited to proving the correctness of DNP programs. In Section 3 we gave a simple correctness proof, taking as a starting point properties of the history functions that we derived from the example program in an informal way. Now we can add rigor to these proofs, because we can derive these functions from our semantics. The sequel to this section is devoted to a description of the derivation of Eqs. (3)–(6) from Section 3.

A good notation is valuable here. The functions f_{set}, f_{merge}, and f_{copy} describe the behavior of pieces of the body of the declaration of the process sort. The corresponding statements are denoted by S_{sort}, S_{merge}, and S_{copy}, which are defined by

```
S_sort  ≡ read(x, unsorted); S_s
S_s     ≡ if x ≥ 0, then S_exp; S_merge else read (y, subsequence)fi; S_copy
S_exp   ≡ expand [sort (unsorted, subsequence1; sorted, empty1 ||
                    keep sort (empty subsequence; subsequence1, empty)]
S_merge ≡ read (y, subsequence);
          while (y ≥ 0 and y ≤ x) do
             write (y, sorted); read (y, subsequence) od;
          write (x, sorted)
S_copy  ≡ while y ≥ 0 do write (y, sorted); read (y, subsequence) od;
          write (−1, sorted).
```

We also need a couple of semantic objects:

$\sigma_0 = \lambda x.undefined$;
$\epsilon_0 = \lambda C.\langle\ \rangle$;
$\bar{\gamma}$ = environment corresponding to the evaluation of the declaration of the program from Section 3, according to clause 4.2.6.

The derivation of f_{copy} from S_{copy} can best be done in three stages:

(1) $\theta_{\text{copy}} = M[\![S_{\text{copy}}]\!]\bar{\gamma}\theta_0$;
(2) $\epsilon_{\text{copy}}(y_0, X, Y) = \theta_{\text{copy}}(\sigma_0[y_0/y])(\epsilon_0[X/\text{unsorted}, Y/\text{subsequence}])$;
(3) $f_{\text{copy}}(y_0, X, Y) = \langle \epsilon_{\text{copy}}(y_0, X, Y)\text{sorted}, \epsilon_{\text{copy}}(y_0, X, Y)\text{empty}\rangle$.

We now derive Eq. (6) from Section 3. To this end we need a little lemma (unwinding the loop):

$$\begin{aligned} &M[\![\text{while } b \text{ do } S \text{ od; } S']\!] \\ &\quad = M[\![\text{if } b \text{ then } S;\ \text{while } b \text{ do } S \text{ od; } S' \text{ else } S' \text{ fi}]\!], \end{aligned}$$

or

$$\begin{aligned} \theta' &= M[\![\text{while } b \text{ do } S \text{ od; } S']\!]\gamma\theta\sigma \\ &= M[\![b]\!]\sigma \rightarrow M[\![S]\!]\gamma\theta'\sigma,\ M[\![S']\!]\gamma\theta\sigma. \end{aligned}$$

Applying this lemma, using 4.2.2.2, and taking $\epsilon' = \epsilon_0[X/\text{unsorted}, Y/\text{subsequence}]$, we get

$$\begin{aligned}\epsilon_{\text{copy}}(y_0, X, Y) = (y \geq 0)\\ &\rightarrow M[\![\text{write}(y, \text{sorted})]\!]\bar{\gamma}\{M[\![\text{read}(y, \text{subsequence})]\!]\bar{\gamma}\theta_{\text{copy}}\}(\sigma_0[y_0/y])\epsilon',\\ &\quad M[\![\text{write}(-1, \text{sorted})]\!]\bar{\gamma}\theta_0(\sigma_0[y_0/y])\epsilon'.\end{aligned}$$

Now suppose $y_0 < 0$. Then (4.2.2.6),

$$\epsilon_{\text{copy}}(y_0, X, \langle z\rangle \hat{\ } Y) = \lambda C.(C \equiv \text{sorted}) \rightarrow \langle -1\rangle, \langle\,\rangle,$$

and thus,

$$f_{\text{copy}}(y_0, X, \langle z\rangle \hat{\ } Y) = (\langle -1\rangle\langle\,\rangle).$$

If $y_0 \geq 0$, we get (4.2.2.6),

$$\epsilon_{\text{copy}}(y_0, X, \langle z\rangle \hat{\ } Y) = \lambda C.(C \equiv \text{sorted}) \rightarrow \langle y_0\rangle \hat{\ }\theta'\text{sorted}, \theta' C,$$

where (using 4.2.2.5),

$$\begin{aligned}\theta' &= M[\![\text{read}(y, \text{subsequence}]\!]\bar{\gamma}\theta_{\text{copy}}(\sigma_0[y_0/y])\\ &\qquad (\epsilon_0[X/\text{unsorted}, \langle z\rangle \hat{\ } Y/\text{subsequence}])\\ &= \theta_{\text{copy}}(\sigma_0[z/y])(\epsilon[X/\text{unsorted}, Y/\text{subsequence}]) = \epsilon_{\text{copy}}(z, X, Y).\end{aligned}$$

This yields

$$f_{\text{copy}}(y_0, X, \langle z\rangle \hat{\ } Y) = (\langle y_0\rangle \hat{\ }(f_{\text{copy}}(z, X, Y) \downarrow 1, f_{\text{copy}}(z, X, Y) \downarrow 2).$$

Thus we have derived Eq. (6) from Section 3.

Derivation of Eq. (5) proceeds similarly, using the definitions

$$\begin{aligned}\theta_{\text{merge}} &= M[\![S_{\text{merge}}; S_{\text{copy}}]\!]\bar{\gamma}\theta_0,\\ \epsilon_{\text{merge}}(x_0, X, Y) &= \theta_{\text{merge}}(\sigma_0[x_0/x])(\epsilon_0[X/\text{unsorted}, Y/\text{subsequence}]),\\ f_{\text{merge}}(x_0, X, Y) &= \langle \epsilon_{\text{merge}}(x_0, X, Y)\text{sorted}, \epsilon_{\text{merge}}(x_0, X, Y)\text{empty}\rangle.\end{aligned}$$

Next, the derivation of Eq. (3). As before, we define

$$\begin{aligned}\theta_{\text{sort}} &= M[\![S_{\text{sort}}]\!]\bar{\gamma}\theta_0,\\ \epsilon_{\text{sort}}(X, Y) &= \theta_{\text{sort}}\sigma_0(\epsilon_0[X/\text{unsorted}, Y/\text{subsequence}]),\\ f_{\text{sort}}(X, Y) &= \langle \epsilon_{\text{sort}}(X, Y)\text{sorted}, \epsilon_{\text{sort}}(X, Y)\text{empty}\rangle.\end{aligned}$$

Applying the clauses from 4.2.2, we derive, taking $\epsilon' = \epsilon_0[X/\text{unsorted}, Y/\text{subsequence}]$,

$$\begin{aligned}&\epsilon_{\text{sort}}(\langle x_0\rangle \hat{\ } X, Y)\\ &\quad = M[\![\text{read}(x, \text{unsorted}); S_s]\!]\bar{\gamma}\theta_0(\epsilon_0[\langle x_0\rangle \hat{\ } X/\text{unsorted}, Y/\text{subsequence}])\\ &\quad = M[\![\text{if } x \geq 0 \text{ then } S_{\text{exp}}; S_{\text{merge}}\\ &\qquad\qquad \text{else read}(y, \text{subsequence})\text{fi}; S_{\text{copy}}]\!]\gamma\theta_0(\sigma_0[x_0/x])\epsilon'\\ &\quad = (x_0 \geq 0) \rightarrow M[\![S_{\text{exp}}]\!]\gamma\theta_{\text{merge}}(\sigma_0[x_0/x])\epsilon',\\ &\qquad\qquad M[\![\text{read}(y, \text{subsequence})]\!]\bar{\gamma}\theta_{\text{copy}}(\sigma_0[x_0/x])\epsilon'.\end{aligned}$$

So we have

$$\begin{aligned}&\epsilon_{\text{sort}}(\langle -1\rangle \hat{\ } X, \langle y_0\rangle \hat{\ } Y)\\ &\quad = \theta_{\text{copy}}(\sigma_0[-1/x, y_0/y])(\epsilon_0[X/\text{unsorted}, Y/\text{subsequence}])\\ &\quad = \epsilon_{\text{copy}}(y_0, X, Y),\end{aligned}$$

which yields (3). (NB. We cheat a little here; we should prove that $\theta_{\text{copy}}\sigma$ does not depend on σx, in order to be able to derive the last equality.)

Finally, Eq. (4). We first investigate what the meaning of "sort" is, as provided by the environment $\bar{\gamma}$. Clause 4.2.6 yields

$$\bar{\gamma} = M[\![\text{sort}(\cdot\cdot) \leftarrow \cdot\cdot]\!](M[\![\text{bottom}(\cdot\cdot) \leftarrow \cdot\cdot]\!](M[\![\text{main}(\cdot\cdot) \leftarrow \cdot\cdot]\!]\bar{\gamma})).$$

According to 4.2.5, we have

$$\bar{\gamma} = \langle \bar{\gamma}_1[\phi_{1s}/\text{sort}, \phi_{1b}/\text{bottom}, \phi_{1m}/\text{main}], \gamma_2[\phi_{2s}/\text{sort}, \phi_{2b}/\text{bottom}, \phi_{2m}/\text{main}]\rangle$$

for the right ϕ_{ij}. In particular, using 4.2.5 again,

$$\bar{\gamma}_1\text{sort} = \phi_{1s} = \phi_{2s}(M[\![S_{\text{sort}}]\!]\bar{\gamma}\theta_0\sigma_0) = \phi_{2s}(\theta_{\text{sort}}\sigma_0) \qquad (*)$$

and also

$$\left.\begin{array}{l}\phi_{2s}\alpha\langle P, Q\rangle\langle R, S\rangle\epsilon R = \alpha(\epsilon_0[\epsilon P/\text{unsorted}, \epsilon Q/\text{subsequence}])\text{sorted} \\ \phi_{2s}\alpha\langle P, Q\rangle\langle R, S\rangle\epsilon S = \alpha(\epsilon_0[\epsilon P/\text{unsorted}, \epsilon Q/\text{subsequence}])\text{subsequence}\end{array}\right. \qquad (**)$$

Now, for $x_0 \geq 0$, using the derivation of ϵ_{sort} as given above, we have

$$\epsilon_{\text{sort}}(x_0\hat{\ }X, Y) = M[\![\text{expand}[B_1 \parallel B_2]\!]\bar{\gamma}\theta_{\text{merge}}(\sigma_0[x_0/x])\epsilon'$$

with

$$\begin{aligned} B_1 &\equiv \text{sort(unsorted, subsequence1; sorted, empty1)} \\ B_2 &\equiv \text{keep sort (empty1, subsequence; subsequence1, empty)} \end{aligned}$$

Next we must find out which function on tuples corresponds to instantiations:

$$\alpha_i = M[\![B_i]\!]\bar{\gamma}\theta_{\text{merge}}(\sigma_0[x_0/x])$$

for $i = 1, 2$. Using 4.2.3, we get

$$\begin{aligned} \alpha_1 &= M[\![B_1]\!]\bar{\gamma}\theta_{\text{merge}}(\sigma_0[x_0/x]) \\ &= \bar{\gamma}_1\text{sort}\ \langle\text{unsorted, subsequence1}\rangle\ \langle\text{sorted, empty1}\rangle. \end{aligned}$$

Using (*) and (**), we see that if α_1 takes the history X as input on channel "unsorted" and history $S1$ on channel "subsequence1" that α_1 will yield on output channel "sorted" the history

$$\begin{aligned} S0 &= \theta_{\text{sort}}\sigma_0(\epsilon_0[X/\text{unsorted}, S1/\text{subsequence}])\ \text{sorted} \\ &= \epsilon_{\text{sort}}(X, S1)\text{sorted} \\ &= (f_{\text{sort}}(X, S1)) \downarrow 1. \end{aligned}$$

Similarly, on output channel "empty1", the history

$$E1 = (f_{\text{sort}}(X, S1)) \downarrow 2$$

will be produced. Furthermore, on all other channels we get $\langle\ \rangle$ as output. Thus the instantiation α_1 corresponds to the tuple function f_{sort} because we have derived that

$$\langle S0, E1\rangle = f_{\text{sort}}(X, S1).$$

Similarly, the history function corresponding to B_2 and α_2 is f_{merge}:

$$\langle S1, E\rangle = f_{\text{merge}}(x_0, E1, Y).$$

Here it is assumed that α_2 takes as input histories $E1$ on channel "empty1" and Y on "subsequence", while it yields histories $S1$ and E on channels "subsequence1" and "empty".

We have now obtained a situation like the one described in Sections 5.2.5 and 5.2.6. A line of reasoning similar to the one we followed there will lead us to the conclusion that the meaning of the corresponding expand statement is a process that takes inputs X and Y on "unsorted" and "subsequence" and yields output histories $S0$ and E on "sorted" and "empty", where $S0$ and E are defined by the smallest solution of

$$\begin{cases} \langle S0,\ E1\rangle = f_{\text{sort}}(X,\ S1) \\ \langle S1,\ E\rangle\ \ = f_{\text{merge}}(x_0,\ E1,\ Y). \end{cases}$$

This immediately yields Eq. (4) in Section 3.

7. CONCLUSIONS AND REMARKS

7.1 Conclusions

In this paper we have given a completely formal definition of DNP, both of its syntax and its semantics. Apart from minor details (e.g., value parameters), all ideas introduced in [11, 12] have been covered. A formal semantics can serve as a test of the informal description of a language. Kahn's language did well in this respect, as we did not find ambiguities. The original papers did have some omissions, however, due to the fact that they introduced some new ideas which were embedded in (a number of variants of) a language that remained sketchy. However, the parts that were not specified precisely could be filled-in in a natural way.

The contributions of our paper are that we have formalized three notions: expansions, containing "keeps" (introduced in passing in [12, Sect. 2.7]); a method to derive history functions from a program text (there is a hint in [11] of McCarthy's method—we have applied the standard techniques from denotational semantics); and a means to define the semantics of recursive expand statements.

The original paper ([11, Sect. 4]); [12] does not deal with formal semantics) was very concise on the last notion. Moreover, recursive expansion is not treated in its full form; the paper only describes nodes that expand immediately (i.e., corresponding in our syntax to a process consisting of only one (expand) statement). An example was given in [11] (see Figure 6), but it does not show the full power of expansion, as it merely builds a "g_2-generator" which yields for all input histories the output μg_2 (whenever g needs input from F, F expands and generates a g-node that provides the input for the original g-node; g therefore yields as output the lub of the chain $g_2\langle\ \rangle \sqsubseteq g_2(g_2\langle\ \rangle) \sqsubseteq \ldots$ which is μg_2). An equivalent nonexpanding network is given in Figure 7 (DUP copies its input to both its output channels. NIL does nothing). Kahn provides the following equations, which should define the meaning of the network in Figure 6:

$$\begin{cases} 0 = F(i) = g_2(F(f(i,\ X))) \\ X = g_1(F(f(i,\ X))) \end{cases} \tag{*}$$

These equations are unclear. For instance, the smallest solution of (*) is $F = (\lambda\tau.\langle\ \rangle)[g_2\langle\ \rangle/i]$ and $X = g_1\langle\ \rangle$, while the intended solution is $F = \lambda\tau.\mu g_2$ and

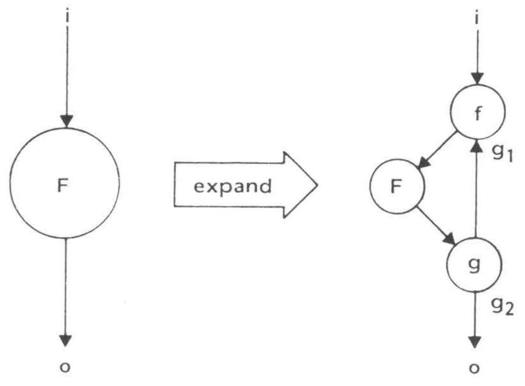

Fig. 6. An example from Kahn's paper.

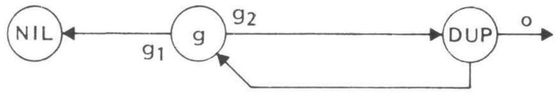

Fig. 7. A nonrecursive network equivalent to the one in Figure 6.

$X = g_1(\mu g_2)$. Apparently, the intended solution is the function F that for all input i yields the smallest output. This function cannot be defined as a simultaneous fixed-point $(F, X) = \mu[\lambda\langle F, X\rangle.\langle \ldots, \ldots\rangle]$, which is suggested by the form of equations (*). Instead it must be defined as an iterated fixed-point expression:

$$F = \mu F[\lambda F.\lambda i.g_2(F(f(i, \mu X[\lambda X.g_1(F(f(i, X)))])))].$$

7.2 Possible Extensions and Topics for Further Research

An interesting idea is to add a bottom element $\perp$ to the set of values V and to turn it into a flat cpo. The value $\perp$ is intended to capture the notion of a nonterminating evaluation of an expression. The first consequence is that the evaluation of a Boolean expression might not terminate also; we must therefore make {true, false}, the set of truth values, a flat cpo by adding a bottom element $\perp$ there too. We must make the (denotation of the) relational operators strict (e.g., $M[\![x = y]\!]\sigma = \perp$ if $M[\![x]\!]\sigma = \perp$ or $M[\![y]\!]\sigma = \perp$).

Σ now turns into a cpo also. If the value of a variable in a state σ equals $\perp$, we declare σ as a whole to be bottom, because the only information needed about such a state is that it is the "result" of a nonterminating computation. So we turn Σ into a flat cpo and we make $\lambda\sigma.\lambda\delta.\sigma]\delta/x]$ strict in σ and δ (i.e., $\perp[\delta/x] = \sigma[\perp/x] = \perp$).

Another consequence is that all continuations that we work with have to be strict: $\theta\perp = \lambda\epsilon.\lambda C.\langle\ \rangle$, for proceeding after we have obtained the "result" of a nonterminating computation should not generate any more output. We therefore redefine the domain of the continuations into $Cont = \Sigma \rightarrow_s Process$ (here $A \rightarrow_s B$ denotes the set of all strict functions from A to B). The functionality of

M has to be redefined as well; it should now be M: *Stat* (resp. *Inst*, *Ndef*) $\rightarrow$ [*Env* $\rightarrow$ [*Cont* $\rightarrow$ ($\Sigma \rightarrow_s$ *Process*)]]. In order to safeguard this functionality, we have to define explicitly that $M[\![\Delta]\!]\gamma\theta\bot$ be $\lambda\epsilon.\lambda C.\langle\,\rangle$ for Δ in *Stat* $\cup$ *Inst* $\cup$ *Ndef*, because this is not guaranteed by the clauses in 4.2.2–4. Note that strictness on a flat domain like Σ implies continuity, and also that 4.2.2.1 now yields the right result if $M[\![b]\!]\sigma = \bot$. If we make the operator $\lambda\alpha.\lambda\beta.\lambda\gamma.\alpha \rightarrow \beta, \gamma$ strict in α, then 4.2.2.3 and 4.2.2.4 will be correct. For instance, if in 4.2.2.3 $M[\![b]\!]\sigma$ yields $\bot$, then the whole construct should yield the bottom element $\lambda C.\langle\,\rangle$ of *Chcont*. Note here that strictness in α implies continuity because $\{\bot$, true, false$\}$ is flat.

How is V^∞ affected? If we look at the semantic definitions we see that the only relevant operations on elements in V^∞ are the test $\epsilon C = \langle\,\rangle$ and the operations *first* and *rest* in 4.2.2.5, and the operation $\lambda\delta.\lambda\tau.\langle\delta\rangle\hat{\ }\tau$ in 4.2.2.6. Equation 4.2.2.5 can be streamlined if we make *first* and *rest* strict (*first* $\langle\,\rangle = \bot$, *rest* $\langle\,\rangle = \langle\,\rangle$):

$$M[\![\text{read }(x, C)]\!]\gamma\theta\sigma\epsilon = \theta(\sigma[\text{first}(\epsilon C)/x])(\epsilon[\text{rest}(\epsilon C)/C]).$$

At first sight, making $\lambda\delta.\lambda\tau.\langle\delta\rangle\hat{\ }\tau$ strict in δ appears to save 4.2.2.6, but this is not so. If the evaluation of the value to be written does not terminate, then there should be no more output, not only on the channel to be written to by the write statement but on all other output channels as well. The solution is to define a strict operator $app(\delta, C, \epsilon)$ ("prefix the history on channel C in ϵ with value δ") by

$$app(\bot, C, \epsilon) = \lambda C.\langle\,\rangle,$$

and for all $\delta \in V$ not equal to $\bot$,

$$app(\delta, C, \epsilon) = \epsilon[\langle\delta\rangle\hat{\ }\epsilon C/C].$$

Now 4.2.2.6 can be rewritten as

$$M[\![\text{write }(t, C)]\!]\gamma\theta\sigma\epsilon = app(M[\![t]\!]\sigma, C, \theta\sigma\epsilon).$$

We conclude that it is possible to accommodate nontermination of the evaluation of expressions in a straightforward way. This is due to the fact that only restricted operations on histories are used; if we would have more complications there, then we have to resort to other solutions, as described for instance in [6].

Next, a few remarks on the value *undefined* $\in V$. Similar techniques can be applied here: a process could flag the use of an uninitialized variable as an error. An appropriate action would be to write the value *undefined* on all output channels and to terminate. Upon receipt of this value, any other process would do the same thing. This can be built into our semantics in the same way as nontermination. Some care has to be exercised if we combine this idea with the addition of $\bot$ to our domains. For instance, the construct *undefined* $\rightarrow \bot, \bot$ should yield *undefined*. We do not investigate this further here.

As the reader might appreciate, giving the formal denotational semantics of a language like DNP is not an easy task. Also, Section 6 makes clear that to give a rigorous formal proof of the properties of DNP programs using this semantics is also not easy. Further research is needed; tools must be built which can be used in such proofs. For instance, we could try to formalize "McCarthy's method," using which the equations in Section 3 were derived from the program text. We could try to build a set of lemmas that could be used to generate such equations.

Our semantics could then be used to prove the lemmas. Another idea is to build a Hoare-like system for DNP (see, for instance, [19] for a proof system for a similar language), or a proof system built on temporal logic, and then to justify such a system using the semantics given here.

Another direction for future research might be to use our semantics to validate and prove the equivalence of various execution methods for DNP programs, for instance, the "coroutine mode" versus the "parallel mode" of execution mentioned in [12, Sect. 3]. Some work has already been done: Kahn mentions [7], more recent work is reported in [2, 3, 8], but it is all based on a subset of DNP (e.g., allowing only static networks with restricted I/O behavior).

7.3 The Relation Between this Paper and Other Work

There are few denotational semantics of languages for parallel programming to be found in the literature that are worked out as completely as this one (the only one we were able to find is [9], dealing with CSP). This is not very surprising: a semantics like ours tends to be complicated, as we have seen, even for a relatively simple language for parallel programming such as DNP, which can be handled using the classical domains from denotational semantics as devised by Scott (see, e.g., [13, 17]).

As soon as parallelism is combined with nondeterminism, more intricate domains are needed. Plotkin [15], Smyth [16], and later de Bakker [4] have worked on this. Such domains have been used to provide the semantics of some notions occurring in languages for parallel programming. For instance, in [4] a small language is defined especially for this purpose, which contains a variant of the CSP communication mechanism. Another way to deal with the complexity of nondeterminism and parallelism is to resort to operational semantics. For instance, in [1] such a semantics is introduced and then used for a justification of a proof system. As far as we know, this paper is the first one in which a fully formal semantics of expansion has been given. Our treatment of parameter passing also seems to be new.

ACKNOWLEDGMENTS

We are grateful to the referees, who provided some stimulating remarks leading to a better paper. We would also like to thank Joke Pannekoek and Debby Musch for typing and retyping our definitely typist-unfriendly manuscript.

REFERENCES

1. APT, K. R. Recursive assertions and parallel programs. *Acta Inf. 15* (1981), 219–232.
2. ARNOLD, A. Sémantique des processus communicants. *RAIRO Theor. Inf. 15*, 2 (1981), 103–139.
3. ARVIND, AND GOSTELOW, K. P. Some relationships between asynchronous interpreters of a data flow language. In *Formal Description of Programming Concepts*, E. J. Neuhold, Ed., North-Holland, Amsterdam, 1978.
4. DE BAKKER, J. W., AND ZUCKER, J. I. Processes and the denotational semantics of concurrency. *Inf. Control 54*, 1/2 (1982), 70–120.
5. BÖHM, A. P. W. Data flow computation. CWI/track nr. 6, Centre for Mathematics and Computer Science, Amsterdam, 1983.
6. DE BRUIN, A. On the existence of Cook semantics. *SIAM J. Comput. 13*, 1 (1984), 1–13.

7. CADIOU, J. M. Recursive definitions of partial functions and their computations. Ph.D. thesis, Stanford Univ., 1972.
8. FAUSTINI, A. A. An operational semantics for pure data flow. In *Automata, Languages and Programming, 9th Colloquium, LNCS 140*, M. Nielsen and E. M. Schmidt, Eds., Springer Verlag, Berlin, 1982, 212–224.
9. FRANCEZ, N., HOARE, C. A. R., LEHMANN, D. J., AND DE ROEVER, W. P. Semantics of nondeterminism, concurrency and communication. *J. Comput. Syst. Sci. 19* (1979), 290–308.
10. GORDON, M. *The Denotational Description of Programming Languages*. Springer Verlag, New York, 1979.
11. KAHN, G. The semantics of a simple language for parallel programming. In *Proceedings IFIP 74*, J. L. Rosenfeld, Ed., North-Holland, Amsterdam, 1974, 471–475.
12. KAHN, G., AND MACQUEEN, D. B. Coroutines and networks of parallel processes. In *Proceedings IFIP 77*, B. Gilchrist, Ed., North-Holland, Amsterdam, 1977, 993–998.
13. MILNE, R., AND STRACHEY, C. *A Theory of Programming Language Semantics*. Chapman and Hall, London, 1977.
14. MORRIS, F. L. The next seven hundred programming language descriptions. Manuscript, Computer Centre, Univ. of Essex, Colchester, 1970.
15. PLOTKIN, G. D. A power domain construction. *SIAM J. Comput. 5*, 3 (Sept. 1976), 452–487.
16. SMYTH, M. B. Power domains. *J. Comput. Syst. Sci. 16* (1978), 23–26.
17. STOY, J. *Denotational Semantics of Programming Languages: The Scott-Strachey Approach*. MIT Press, Cambridge, Mass., 1977.
18. STRACHEY, C., AND WADSWORTH, C. Continuations: A mathematical semantics for handling full jumps. Tech. Mon. PRG-11, Oxford Univ. Computing Lab., Programming Research Group, 1974.
19. ZWIERS, J., DE BRUIN, A., AND DE ROEVER, W. P. A proof system for partial correctness of dynamic networks of processes. In *Proceedings of Logic of Programs, 1983, LNCS 164*, Springer Verlag, Berlin, 513–527.

Received November 1984; revised June 1985; accepted July 1985

SAMENVATTING

In essentie bestaat dit proefschrift uit vier artikelen, die gereproduceerd zijn in de hoofdstukken 2 t/m 5. Deze hoofdstukken behandelen een aantal onderwerpen op het terrein van de semantiek van programmeertalen. In de artikelen worden een aantal programmeerconcepten onderzocht door er ofwel een denotationele semantiek voor te konstrueren ofwel, als de betekenis ervan is vastgesteld via een ander formalisme, deze te onderzoeken met behulp van denotationele semantiek of technieken eruit. Een gemeenschappelijk kenmerk van deze artikelen is dat de concepten die ik onderzocht heb een struktuur hebben die uitnodigt tot het gebruik van continuaties. Dit is waarschijnlijk het eenvoudigst uit te leggen door enkele alinea's te wijden aan de inhoud van de artikelen.

In hoofdstuk 2 wordt een semantiek in de stijl van Cook bestudeerd. De betekenis van de diverse programmakonstrukties wordt volgens deze semantiek vastgelegd door een aantal rekursieve vergelijkingen, een aanpak die ook gebruikelijk is in de denotationele semantiek. Een probleem is echter dat de standaardtechnieken uit de denotationele semantiek hier niet werken, omdat de operatoren geinduceerd door dit soort vergelijkingen niet krachtig genoeg (dat wil zeggen: kontinu) zijn. In mijn artikel blijkt dat na zekere aanpassingen deze technieken toch gebruikt kunnen worden, waarmee dan definities in de stijl van Cook gerechtvaardigd zijn.

Hoofdstuk 3 geeft twee semantieken, een denotationele en een operationele, voor de kern van de programmeertaal SNOBOL4, te weten het patroonherkenningsalgoritme. Dit algoritme werkt volgens het pricipe "gissen en vergissen" (trial and error). Het idee is dat er een aantal mogelijkheden onderzocht dient te worden en dat gebeurt door in eerste instantie een ervan te bekijken. Als blijkt dat deze mogelijkheid niet voldoet, dan wordt een nieuwe mogelijkheid geprobeerd door die situatie (min of meer) te herstellen waar voor het laatst een keuze bestond uit een aantal nog te onderzoeken alternatieven, en vervolgens een nieuwe keuze te maken.

Dit is een stijl van programma-uitvoering die nogal afwijkt van de stijl die voorgestaan wordt door de voorstanders van gestruktureerd programmeren. Een programmafragment is niet meer een module met een enkele ingang en een uitgang, die samengevoegd kan worden met andere modules met dezelfde eigenschappen tot grotere modules van hetzelfde type. Integendeel, tijdens verwerking van een bepaald gedeelte van een programma kan blijken, dat er ooit in een heel ander fragment een verkeerde keuze is gedaan, zodat verwerking van het programma terugspringt naar dat keuzepunt. Dat komt erop neer dat, naast de standaardingang waar verwerking

van een programmagedeelte begint, tijdens verwerking ervan een aantal nieuwe ingangen gekreëerd worden, die gebruikt kunnen worden als in een later stadium de berekening faalt in een heel ander gedeelte van het programma.

Dit soort programma-exekutie kan niet eenvoudig beschreven worden met direkte denotationele semantiek, een stijl die wel goed toepasbaar is voor meer gestruktureerde taalkonstrukties. Hier zal gebruik gemaakt moeten worden van continuaties, ofwel voortzettingen. De betekenis van een instruktie uit een programmeertaal is nu een funktie die de beginsituatie (zeg de inhoud van het geheugen van de computer) voordat de instruktie uitgevoerd zal worden, zal omzetten in een eindantwoord (zeg de output van het programma), waarbij inbegrepen is het effekt van het uitvoeren van de rest van het programma, dat geëxekuteerd zal worden als de onderhavige instruktie afgehandeld is. Deze toekomst van de berekening kan niet afgelezen worden uit de instruktie onder behandeling, en daarom wordt bij het bepalen van de betekenis van de instruktie deze toekomst meegegeven als parameter. Zo'n parameter nu is een continuatie, een voortzetting die voor elke mogelijke situatie direkt na evaluatie van de instruktie beschrijft welk eindantwoord het uitvoeren van de rest van het programma zal opleveren.

"Gissen en vergissen" kan gevangen worden door de betekenis van een instruktie afhankelijk te laten zijn van twee continuaties als parameter, te weten een "succesvoortzetting" die het gedeelte van het programma beschrijft dat uitgevoerd wordt als verwerking van de instruktie normaal termineert, en een "faal-voortzetting" die aangeeft wat er gebeurt als tijdens verwerking van de instruktie blijkt dat een keuze die ooit gemaakt is onterecht geweest is.

Continuaties worden gebruikt in een sektie van hoofdstuk 2 en ook in de hoofdstukken 4 en 5. Hoofdstuk 4 gaat over het **goto** statement, de standaard sprongopdracht. Een van de bezwaren die tegen deze instruktie ingebracht wordt is dat taalfragmenten nu meerdere ingangen en uitgangen kunnen hebben: ieder label dat gedefinieerd wordt in het fragment dat van buitenaf toegankelijk is legt een nieuwe ingang vast en als het fragment sprongopdrachten bevat naar labels die niet in het fragment zelf gedefinieerd zijn, dan is er meer dan een uitgang. De konklusie dat dat er geen nette bewijzen te leveren zijn van programma's met **goto**'s is evenwel niet terecht. Er zijn een aantal korrekte formele bewijssystemen voorgesteld en van twee daarvan wordt in hoofdstuk 4 bewezen dat ze gezond en (relatief) volledig zijn. Voor deze rechtvaardiging gebruik ik denotationele semantieken en het blijkt dat gebruik gemaakt kan worden van zowel direkte semantiek als continuatiesemantiek.

Het laatste hoofdstuk behandelt een taal voor parallel programmeren, gebaseerd op de ideeën van Kahn, DNP genaamd (dynamic networks of processes). In Kahn's artikelen wordt een globale omschrijving gegeven van de achterliggende principes en er worden een aantal voorbeelden behandeld, maar er wordt niet expliciet een programmeertaal gedefinieerd, laat staan dat er een formele denotationele semantiek gegeven wordt. Dat laatste wordt gerealiseerd in hoofdstuk 5. De semantiek die daar gedefinieerd wordt, maakt nadrukkelijk gebruik van continuaties.

Het eerste hoofdstuk van dit proefschrift bevat een inleiding in de begrippen die in de latere hoofdstukken aan de orde zullen komen. Zo wordt bijvoorbeeld het begrip continuatie degelijker geintroduceerd dan hier mogelijk is.

Daarnaast wordt in sektie 1.3 ingegaan op de vraag in hoeverre het gebruik van continuaties in de andere hoofdstukken nodig en zinnig is, en of er geen direkte denotationele semantiek gegeven kan worden voor de daar behandelde concepten.

De konklusie is dat dat in de meeste gevallen wel mogelijk is, maar dat een direkte semantiek voor het soort programmeerconcepten dat in dit proefschrift behandeld wordt, al snel tot onoverzichtelijke definities leidt, voornamelijk omdat er rekening gehouden moet worden met veel details. Dat laatste blijkt bij continuatiesemantiek aanzienlijk minder het geval te zijn.

CURRICULUM VITAE

Naam	Arie de Bruin
Geboren	27 april 1949 te 's-Gravenhage

1967	Eindexamen gymnasium β, Chr. Lyceum "Populierstraat" te 's-Gravenhage
1972	Kandidaatsexamen Wis- en Natuurkunde, cum laude, Vrije Universiteit Amsterdam
1972–1973	Assistent Mathematisch Centrum Amsterdam, afd. informatica
1977	Doctoraalexamen Wiskunde met groot bijvak Informatica, cum laude, afstudeerhoogleraar prof. dr R.P. van de Riet
1977–1981	Wetenschappelijk medewerker Mathematisch Centrum Amsterdam, afd. informatica
1981–nu	Wetenschappelijk medewerker informatica, Erasmus Universiteit Rotterdam

CURRENT ADDRESS:
Erasmus Universiteit Rotterdam
Faculty of Economics
P.O. Box 1738
3000 DR ROTTERDAM
The Netherlands