Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

MATHEMATICAL CENTRE TRACTS 131

CORRECTNESS PRESERVING PROGRAM REFINEMENTS: PROOF THEORY AND APPLICATIONS

R.J.R. BACK

MATHEMATISCH CENTRUM AMSTERDAM 1980

1980 Mathematics subject classification: 68B10

ACM-Computing Reviews-category: 5.21, 5.24

ISBN 90 6196 207 2

To my parents,
with love
and admiration.

ACKNOWLEDGEMENTS

This tract is a revised version of my thesis, which originally appeared as [1]. I have been helped by several people, to whom I want to express my gratitude. I am most indebted to Maaret Karttunen, for the many discussions and arguments in which the ideas of the thesis were clarified. Reino Kurki-Suonio and Jaco de Bakker gave me the encouragement and support necessary to get, first the thesis and then this revision done. I have profited greatly from discussions with, and the support of, Ole-Johan Dahl, Kai Koskimies, Lambert Meertens, Juha Oikkonen, Martti Tienari, J.V. Tucker and Esko Ukkonon. The excellent typing by Linda Brown speaks for itself.

I would particularly like to thank Lars Backström and the Computing Centre of the University of Helsinki for their generosity in giving me plenty of time to concentrate on my work, in a warm and friendly atmosphere. I am also indebted to the Mathematical Centre, Amsterdam, for the opportunity to work in its scientifically stimulating and pleasant environment. The financial support given by Magnus Ehrnroth Foundation and Svenska Vetenskapliga Centralrådet is gratefully acknowledged.

Finally, I wish to thank my wife Barbro and my children Pia and Rasmus for reminding me that there are more important things in life than writing publications like this.

CONTENTS

CHAPTER 1

INTRODUCTION

The *stepwise refinement* method, developed primarily by DIJKSTRA [13,14, 15] and WIRTH [43,44], is nowadays an important and well-established program construction technique. The basic idea is that a program should be constructed by a sequence of refinement steps, leading from an initial specification to the final program. Each refinement step results in a new version of the program, usually improving on the previous version in some respect. It can, for example, make less severe assumptions about the basic operations and/or data types available, or it can be more efficient than the previous version.

Stepwise refinement was originally proposed by DIJKSTRA [13] as a *constructive* approach to program proving. According to this view, if each refinement step is very carefully carried out, so that it can be seen to preserve the correctness of the previous version of the program, then the final program must be correct *by construction*. In practice, however, the refinement steps made are often far from trivial, therefore making it difficult to judge the correctness of a refinement step on a purely intuitive basis. Examples of such nontrivial refinement steps include procedure and data type implementations, changes made in the data or control structures of the program, as well as applications of general program transformation rules.

We consider here the problem of how the correctness of refinement steps can be shown. A formal system will be presented in which correctness of refinement steps can be *proved,* thus providing a rigorous foundation for the use of stepwise refinement as a constructive proof technique for program correctness.

The approach that we will take here is best characterized by listing some of the more important goals that we have tried to achieve.

(1) We wanted to stay as close as possible to the way in which stepwise refinement is used by Dijkstra and Wirth in the references cited above. We especially wanted to keep the open-ended nature of their method, where

any kind of refinement step is allowed, as long as it can be seen to preserve the correctness of the preceding version.

(2) We wanted to treat refinement in a broad sense, including not only implementation of procedures but also refinements concerned with data representations and control structures of a program, as well as the use of program transformation rules.

(3) We also wanted to keep our programming language as simple as possible. In particular, this meant that we did not want to introduce such complicated constructs as procedures or abstract data types into our language.

(4) We wanted to reason about the correctness of a refinement step in a formal system, with a fixed set of axioms and proof rules, and not base our proofs on semantic arguments.

(5) We did not want to invent a formal system of ouw own, but rather wanted to use an existing system with well-known mathematical properties.

(6) Finally, we decided to consider only the total correctness of programs, leaving partial correctness and other possible correctness criteria aside.

These goals serve to distinguish our approach from other approaches to program proving. Thus the axiomatic technique by HOARE [23,24,25] agrees with point (2) above, except for program transformations (which are treated in his style in GERHART [20]), and also with (4) , but only partially with (1) and (3) and not at all with (5) and (6). HAREL and al [22] extend Hoare's technique in the direction we are interested in, treating also total correctness of programs, but otherwise the same comments hold for their system.

On the other hand, the weakest precondition technique used by DIJKSTRA [15] does not agree with (2) and only partially with (4), but otherwise is quite close to the approach taken here. Our work is essentially an extension and a formalization of Dijkstra's weakest precondition technique.

The work that has been done on program transformation systems, such as that by GERHART [20], BURSTALL and DARLINGTON [10], WEGBREIT [42] and LOVEMAN [33], has concentrated more on finding useful program transformation rules, pushing the formal aspects of the method somewhat into the background. The same applies to the treatment of stepwise refinement in general given in KNUTH [30] and CORRELL [11].

Program transformation systems have been considered from a more formal point of view in the project CIP [6,7]. The work by BROY, GNATZ and WIRSING [9] in this project is in some respects close to ours, although their

approach is much more oriented towards semantics than ours. Also, the basic concept of refinement used by these authors is different from the one used here (the work reported here was done independently of these and is based on the authors' thesis [1]). Another approach in the same direction as ours is described in MEERTENS [35].

Goals (4), (5) and (6) lead quite naturally to the use of the weakest precondition technique. It turns out that the most appropriate formal system for expressing the weakest preconditions of programs and reasoning about them is the infinitary logic $L_{\omega_1\omega}$. This is an extension of the usual first-order logic, in which infinite disjunctions and conjunctions over formulas are allowed. The infinite disjunctions are needed for expressing the weakest preconditions of loops. We will make extensive use of this logic in this tract, therefore devoting the first main chapter to an exposition of the syntax, semantics and axiomatization of this logic.

Goals (2) and (3) are in potential conflict with each other. The way out of this dilemma is to design the language in such a way that the *effect* of operational and representational abstraction in program development can be achieved, even if the language does not permit the explicit declaration of these constructs. The solution given will be quite close to the way in which operational and representational abstraction is used in DIJKSTRA [15], thus also agreeing with goal (1).

The programming language to be used will contain a new kind of primitive statement called an *atomic description*. It can be loosely characterized as a nondeterministic assignment statement with an associated change of scope (i.e. a change of the set of variables available). The *descriptions* will be constructed out of the atomic descriptions using control structures such as composition, selection, iteration and nondeterministic binary choice. It will be possible to express both programs and their specifications in this language, therefore making it unnecessary to consider two different languages as is usually done, a specification language on the one hand and a programming language on the other. We will devote Chapter 3 to explaining the syntax and semantics of this language.

Goal (1) and, in particular, the open-endedness of stepwise refinement have been achieved by introducing correctness of refinement as a binary relation between descriptions. Thus $S \leq S'$ expresses the fact that the description S' is a correct refinement of the description S. This *refinement relation* will be reflexive and transitive, justifying the stepwise method of program construction. Thus if

$$S_0, S_1, \ldots, S_{n-1}, S_n$$

is the sequence of program versions constructed, with S_0 as the initial specification and S_n as the final program, such that

$$S_i \leq S_{i+1}, \qquad i = 0,1,\ldots,n-1,$$

then

$$S_0 \leq S_n,$$

i.e. the final program S_n is a correct refinement of the specification S_0.

The refinement relation will be defined in Chapter 4, where we show some simple properties of this relation. In the same chapter we also introduce an equivalence relation between descriptions, obtained by requiring mutual refinement between them. Again in Chapter 4 we give an important characterization of refinement using weakest preconditions, on which the technique for proving the correctness of a refinement step will be based.

The general proof rule for correct refinement between descriptions will be given in Chapter 5. Proving $S \leq S'$ for descriptions S and S' will essentially amount to computing the weakest preconditions for these and proving a specific formula of $L_{\omega_1\omega}$ involving the weakest preconditions. This proof rule will be complete in the sense that $S \leq S'$ will hold if and only if the corresponding formula of $L_{\omega_1\omega}$ is provable.

In Chapter 6 we go on to show how stepwise refinement can be carried out using descriptions. We will show how to model top-down program development, operational and representational abstraction and how to justify the use of program transformation rules. (For those readers who are not familiar with the stepwise refinement technique, we recommend a glance at Section 6.1 of this chapter, where an example is given.)

Descriptions describe nondeterministic state transformations of *unbounded nondeterminism*. This gives some problems in expressing weakest preconditions, as discussed in [15]. To avoid these problems, we will restrict ourselves to refinements between a certain kind of descriptions only, called *program descriptions*. These are not as general as descriptions, but are more convenient to work with. Programs and program specifications will be special

kinds of program descriptions.

Finally, in Chapter 7 we give an example of formal program development using program description. We will give special proof rules for handling commonly occurring refinement steps, such as procedure implementations, introducing assertions into programs, handling representational abstraction and changing the control structure of a program. These special proof rules will all be derived from the general proof rule for refinement using the axioms and inference rules of $L_{\omega_1\omega}$, thereby showing the suitability of this logic for reasoning about programs and the generality of the proof rule for refinement.

CHAPTER 2

THE INFINITARY LOGIC $L_{\omega_1\omega}$

We will choose an infinitary logic called $L_{\omega_1\omega}$ as the underlying logic for carrying out proofs of program properties. This logic is an extension of ordinary first-order logic, allowing disjunctions and conjunctions over a countably infinite number of formulas. To handle these infinite disjunctions and conjunctions, we need inference rules with a countably infinite number of premises, which in turn forces us to accept infinitely long (but countable) proofs.

The need for infinite disjunctions arises in connection with the proof rule for loops. The assertion that a loop in a deterministic program terminates correctly, for a given set of initial states, can be expressed as an infinite disjunction in the following way: for every initial state in the given set the loop either terminates correctly without any iterations, or it terminates correctly after one iteration, or ..., or it terminates correctly after n iterations, or ... . If the set of initial states given is infinite, then it will not, in general, be possible to give an upper bound N such that the loop will terminate for any initial state in the set after at most N iteration. Hence the disjunction must contain an infinite number of subassertions.

The logic $L_{\omega_1\omega}$ is a special case of a general class of infinitary logics, whose members are denoted $L_{\alpha\beta}$. The logic $L_{\alpha\beta}$ is like ordinary first-order logic, except that it allows disjunctions and conjunctions over fewer than α formulas, and universal and existential quantification over variable sequences with fewer than β variables, where α and β are two infinite cardinal numbers, $\beta \leq \alpha$. By choosing $\alpha = \omega_1$ and $\beta = \omega$, we get $L_{\omega_1\omega}$, in which we allow disjunctions and conjunctions over countable sets of formulas, but quantification only over finite sequences of variables. (ω is the cardinality of the set of natural numbers, while ω_1 is the next bigger cardinal number. Thus $\alpha < \omega_1$ means that α is a countable ordinal number, while $\alpha < \omega$ means that α is a finite ordinal number.) If we choose $\alpha = \beta = \omega$, we get the usual first-

order logic, in which only finite disjunction, conjunction and quantification is allowed.

Our treatment of $L_{\omega_1\omega}$ below is based on KARP [27], with some changes in the notation. The treatment is self-contained, except that proofs of the lemmas are omitted. The lemmas follow quite straightforwardly from the basic theorems proved by Karp. The logic $L_{\omega_1\omega}$ is also treated in SCOTT [40], FEFERMAN [19] and KEISLER [29], just to mention a few. We have chosen KARP [27] as our basis because it uses a familiar Hilbert-like proof theoretic approach to this logic.

2.1. THE SYNTAX OF $L_{\omega_1\omega}$

Every $L_{\omega_1\omega}$ *language* L has the same set of *logical symbols*

$$\sim \quad \Rightarrow \quad \wedge \quad \forall \quad = \quad , \quad (\ )$$

and the same set of *variables*

$$v_0, v_1, \ldots, v_\xi, \ldots, \qquad \xi < \omega_1.$$

A particular $L_{\omega_1\omega}$ language L is characterized by its *non-logical* symbols. These are of three kinds. We have the *constant symbols*

$$c_0, c_1, \ldots, c_\xi, \ldots, \qquad \xi < \delta_1, \ \delta_1 < \omega_1$$

and for each n, $0 < n < \omega$, the n-*place function symbols*

$$F_0^n, F_1^n, \ldots, F_\xi^n, \ldots, \qquad \xi < \delta_2, \ \delta_2 < \omega_1$$

and the n-*place predicate symbols*

$$G_0^n, G_1^n, \ldots, G_\xi^n, \ldots, \qquad \xi < \delta_3, \ \delta_3 < \omega_1.$$

If L and L' are two $L_{\omega_1\omega}$ languages, such that each non-logical symbol of L is a non-logical symbol of L', then L' is said to be an *expansion* of L.

Let L be an $L_{\omega_1\omega}$ language. The *terms* of L are defined as usual:

(i) Each variable is a term of L.

(ii) Each constant symbol of L is a term of L.

(iii) If $t_1,\ldots,t_k$ are terms of L and F is a k-place function symbol, then $F(t_1,\ldots,t_k)$ is a term of L.

To be more precise, we should define the set of terms of L as the least set containing the variables and the constants of L and closed under rule (iii). The inductive definitions given here should be understood in this way, i.e. an element belongs to an inductively defined set if and only if it can be seen to belong to the set by the rules given for defining the set.

The *formulas* of L are defined as follows:

(i) If t_1 and t_2 are terms of L, then $t_1 = t_2$ is a formula of L.

(ii) If $t_1,\ldots,t_k$ are terms of L and G is a k-place predicate symbol of L, then $G(t_1,\ldots,t_k)$ is a formula of L.

(iii) If A_0 is a formula of L, then $(\sim A_0)$ is a formula of L.

(iv) If A_0 and A_1 are formulas of L, then $(A_0 \Rightarrow A_1)$ is a formula of L.

(v) If $0 < \delta < \omega_1$ and A_ξ is a formal of L for $\xi < \delta$, then $(\wedge_{\xi<\delta} A_\xi)$ is a formula of L.

(vi) If v is a finite nonempty sequence of variables and A_0 is a formula of L, then $(\forall v A_0)$ is a formula of L.

The formula $(\wedge_{\xi<\delta} A_\xi)$ is a shorthand for the formula $(\wedge A_0 \ldots A_\xi \ldots)$, where $A_0 \ldots A_\xi \ldots$ is a (possibly infinite) sequence of formulas A_ξ, $\xi < \delta$. In KARP [64] infinitely long sequence of this kind are given rigorous treatment. We will here rely on the intuitive notion of an infinite sequence of formulas, referring to KARP [64] for a formal definition of the concepts presented here.

The other connectives and quantifiers are introduced as abbreviations in the usual way:

(i) $(A_0 \wedge A_1)$ stands for $(\wedge_{\xi<2} A_\xi)$,

(ii) $(\vee_{\xi<\delta} A_\xi)$ stands for $(\sim \wedge_{\xi<\delta} (\sim A_\xi))$, $\delta < \omega_1$,

(iii) $(A_0 \vee A_1)$ stands for $(\vee_{\xi<2} A_\xi)$,

(iv) $(A_0 \Leftrightarrow A_1)$ stands for $((A_0 \Rightarrow A_1) \wedge (A_1 \Rightarrow A_0))$ and

(v) $(\exists v A_0)$ stands for $(\sim \forall v (\sim A_0))$.

An *occurrence* of a variable v_ξ in a formula is said to be *bound*, if the occurrence is within a subformula of the form $(\forall v\ A')$, where v_ξ is one of the variables of v. We say that an *occurrence* of a variable v_ξ in a formula is *free*, if this occurrence is not bound. The *variable* v_ξ is said to be *free* in a formula, if there is a free occurrence of the variable v_ξ in the formula.

Similarly, the *variable* v_ξ is said to be *bound* in a formula, if there is a bound occurrence of the variable v_ξ in the formula.

Let $t_1,\ldots,t_k$ be terms of L, and let $x_1,\ldots,x_k$ be *distinct* variables, i.e. for each i, j such that $1 \le i < j \le k$, $x_i \neq x_j$. Let t be a term of L. Then $t[t_1/x_1,\ldots,t_k/x_k]$ denotes the term of L obtained by substituting simultaneously for $i = 1,\ldots,k$ the term t_i for each occurrence of x_i in t.

If A is a formula of L and t is a term of L, then t is said to be *free for the variable* v_ξ in A, if no free occurrence of v_ξ in A is an occurrence in a subformula $(\forall v A')$ of A, where v contains a variable that occurs in t.

Let $t_1,\ldots,t_k$ be terms of L and $x_1,\ldots,x_k$ be distinct variables. Let A be a formula of L. Then $A[t_1/x_1,\ldots,t_k/x_k]$ denotes the formula of L that we obtain by first changing the variables bound in A so that each term t_i will be free for x_i in A, and then substituting simultaneously for $i = 1,\ldots,k$ the term t_i for each free occurrence of x_i in A. The replacement of bound variables with new variables is assumed to be done in a systematic fashion, so that the formula $A[t_1/x_1,\ldots,t_k/x_k]$ is uniquely defined.

A formula of L that does not contain any free variables is called a *sentence*.

2.2. THE SEMANTICS OF $L_{\omega_1\omega}$

Let Tr be the set of *truth values*, $Tr = \{tt,ff\}$. Here tt stands for "true" and ff stands for "false". A k-*place predicate on the set* D is a function from D^k to Tr, assigning a truth value to each k-tuple of D.

A *structure* for the $L_{\omega_1\omega}$ language L is a pair $M = \langle D,I\rangle$, where D is a nonempty set and I is a function that assigns to each constant symbol of L an element in D, to each k-place function symbol of L a k-place function in D and to each k-place predicate symbol of L a k-place predicate on D.

Let V be a nonempty set of variables. A V-*assignment* in D is a function $s: V \to D$. The set of all V-assignements in D is denoted D^V. Given a V-assignment s in D, the distinct variables $x_1,\ldots,x_k$ and the elements $a_1,\ldots,a_k$ of D (not necessarily distinct), $s\langle a_1/x_1,\ldots,a_k/x_k\rangle$ denotes the V'-assignment s' in D where $V' = V \cup \{x_1,\ldots,x_k\}$ and $s'(x_i) = a_i$ for $i = 1,\ldots,k$, while $s'(v_\xi) = s(v_\xi)$ for each $v_\xi \in V$, $v_\xi \neq x_i$ for $i = 1,\ldots,k$.

Let $M = \langle D,I\rangle$ be a structure for L. Let t be a term of L, and let V be a set of variables such that any variable occurring in t belongs to V. We define the *value of* t *in* M *for the* V-*assignment* s, denoted $val_M(t,s)$, as follows:

(i) If t is the variable v_ξ in V, then $val_M(t,s) = s(v_\xi)$.

(ii) If t is the constant symbol c_ξ, then $val_M(t,s) = I(c_\xi)$.

(iii) If t is the term $F(t_1,\ldots,t_k)$, where F is a k-place function symbol, then $val_M(t,s) = I(F)(val_M(t_1,s),\ldots,val_M(t_k,s))$.

Similarly, we define the *value of the formula* A *in* M *for the* V*-assignment* s, when each free variable of A is in V, to be an element of Tr, denoted $val_M(A,s)$:

(i) If A is $t_1 = t_2$, then $val_M(A,s) = tt$ iff $val_M(t_1,s) = val_M(t_2,s)$.

(ii) If A is $G(t_1,\ldots,t_k)$, where G is a k-place predicate symbol, then $val_M(A,s) = I(G)(val_M(t_1,s),\ldots,val_M(t_k,s))$.

(iii) If A is $(\sim A_0)$, then $val_M(A,s) = tt$ iff $val_M(A_0,s) = ff$.

(iv) If A is $(A_0 \Rightarrow A_1)$, then $val_M(A,s) = tt$ iff $val_M(A_0,s) = ff$ or $val_M(A_1,s) = tt$.

(v) If A is $(\wedge_{\xi<\delta} A_\xi)$, then $val_M(A,s) = tt$ iff $val_M(A_\xi,s) = tt$ for each $\xi < \delta$.

(vi) If A is $(\forall v A_0)$, then $val_M(A,s) = tt$ iff $val_M(A_0, s\langle a_1/x_1,\ldots,a_k/x_k\rangle) = tt$ for every $\langle a_1,\ldots,a_k\rangle \in D^k$, where $x_1,\ldots,x_k$ are the distinct variables occurring in v.

LEMMA 2.1. *Let* s *be a* V*-assignment in* D *and let* s' *be a* V'*-assignment in* D. *If both* V *and* V' *contain each variable occurring in the term* t, *and if* $s(v_\xi) = s'(v_\xi)$ *for each such variable* v_ξ *in* t, *then* $val_M(t,s) = val_M(t,s')$. *Similarly, if both* V *and* V' *contain each variable occurring free in the formula* A, *and* $s(v_\xi) = s'(v_\xi)$ *for each such free variable* v_ξ, *then* $val_M(A,s) = val_M(A,s')$.

PROOF. Theorems 3.5.5(i) and 9.1.5 in KARP [27]. □

We say that the formula A *holds* in the structure $M = \langle D,I\rangle$, if for some set V of variables containing all the variables free in A, we have $val_M(A,s) = tt$ for every V-assignment s in D. By the lemma above, the specific choice of V does not affect the property that a formula holds in M.

We say that a structure M is a *model* for a set of formulas Δ, if each formula of Δ holds in M. The formula A is said to be a *semantic consequence* of Δ, denoted $\Delta \models A$, if A holds in every model of Δ. A formula A is said to be *valid* if it is a semantic consequence of the empty set of formulas.

Let L' be an expansion of the language L, and let $M = \langle D,I\rangle$ be a structure for L. A structure $M' = \langle D,I'\rangle$ for L' where I' agrees with I on the nonlogical symbols of L is said to be an *expansion of* M *to* L'.

2.3. PROOFS IN $L_{\omega_1\omega}$

KARP [27] gives the following axiom system for an $L_{\omega_1\omega}$ language L. The *axioms* are:

I1. $(A_0 \Rightarrow (A_1 \Rightarrow A_0))$

I2. $((A_0 \Rightarrow (A_1 \Rightarrow A_2)) \Rightarrow ((A_0 \Rightarrow A_1) \Rightarrow (A_0 \Rightarrow A_2)))$

N1. $((\sim A_0 \Rightarrow \sim A_1) \Rightarrow (A_1 \Rightarrow A_0))$

C1. $(\wedge_{\xi<\delta} (A_\delta \Rightarrow A_\xi) \Rightarrow (A_\delta \Rightarrow \wedge_{\xi<\delta} A_\xi)), \quad 0 < \delta < \omega_1$

C2. $(\wedge_{\xi<\delta} A_\xi \Rightarrow A_\eta), \quad \eta < \delta, \ 0 < \delta < \omega_1$

Q1. $(\forall v(A_0 \Rightarrow A_1) \Rightarrow (A_0 \Rightarrow \forall v A_1))$, if no variable of v is free in A_0

Q2. $(\forall v A_0 \Rightarrow A_0[t_1/x_1,\ldots,t_k/x_k])$, where $x_1,\ldots,x_k$ are the distinct variables of v

E1. $t_1 = t_1$

E2. $(\wedge_{i\leq k} (t_i = t_i') \Rightarrow F(t_0,\ldots,t_k) = F(t_0',\ldots,t_k'))$

E3. $(\wedge_{i\leq k} (t_i = t_i') \Rightarrow G(t_0,\ldots,t_k) \Rightarrow G(t_0',\ldots,t_k'))$.

The *inference rules* are:

MP. $\dfrac{A_0,\ (A_0 \Rightarrow A_1)}{A_1}$

CN. $\dfrac{A_0,\ldots,A_\xi,\ldots,\ \xi<\delta}{\wedge_{\xi<\delta} A_\xi}, \quad 0 < \delta < \omega_1$

GN. $\dfrac{A_0}{\forall v A_0}$

Here $A_0,\ldots,A_\xi,\ldots$ are formulas of L, $t_0,\ldots,t_k,t_0',\ldots,t_k'$ are terms of L, F a k+1-place function symbol of L, G a k+1-place predicate symbol of L and v is a nonempty sequence of variables.

A *proof* in L of the formula A from the set of formulas Δ is a sequence

$$B_0,\ldots,B_\xi,\ldots,B_\eta$$

of formulas of L, where $\eta < \omega_1$, $A = B_\eta$, and for each $\xi \leq \eta$, B_ξ is either an axiom, a formula of Δ or has been obtained from previous formulas in the sequence by applying one of the inference rules. A formula A is *provable from*

Δ, denoted $\Delta \vdash A$, if there is a proof of A from Δ. The formula A is a *theorem*, denoted $\vdash A$, if it is provable from the empty set of formulas.

The logic $L_{\omega_1\omega}$ is similar to first-order logic in that it is complete, in the following sense (KARP [27]):

LEMMA 2.2. (Completeness). *For any formula* A *of* L *and any countable set of sentences* Δ *of* L, $\Delta \models A$ *if and only if* $\Delta \vdash A$.

PROOF. Follows from Theorem 11.2.4 and 11.4.1 in KARP [27]. □

The following results will be useful later. The proofs of these are straightforward consequences of the theorems proved in KARP [27]. We assume in the lemmas that Δ is a set of sentences of L.

LEMMA 2.3. (Deduction theorem). *Let* A *and* B *be two formulas of* L, *where the free variables of* A *are* $x_1,\ldots,x_k$. *Let* L' *be the expansion of* L *that we get by adding the new constant symbols* $d_1,\ldots,d_k$ *to* L. *Then* $\Delta \vdash A \Rightarrow B$ *in* L, *if* $\Delta \cup \{A[d_1/x_1,\ldots,d_k/x_k]\} \vdash B[d_1/x_1,\ldots,d_k/x_k]$ *in* L'.

PROOF. Follows from the Theorems 11.2.4 and 11.3.1 of KARP [27]. □

LEMMA 2.4. (Inference rule for disjunction). *If* $\Delta \vdash A_\xi \Rightarrow B$ *for* $\xi < \delta$, $\delta < \omega_1$ *then* $\Delta \vdash \bigvee_{\xi<\delta} A_\xi \Rightarrow B$.

PROOF. Follows from the definition of disjunction, using axiom C1 and Theorem 11.2.3(ii) in KARP [27]. □

LEMMA 2.5. (Axiom for disjunction). $\Delta \vdash A_\eta \Rightarrow \bigvee_{\xi<\delta} A_\xi$, *for* $\eta < \delta$, $\delta < \omega_1$.

PROOF. Follows from the definition of disjunction, using axiom C2. □

LEMMA 2.6.

$$\vdash A[t_1/x_1,\ldots,t_k/x_k] \iff \forall x_1\ldots x_k(x_1 = t_1 \wedge \ldots \wedge x_k = t_k \Rightarrow A)$$

and

$$\vdash A[t_1/x_1,\ldots,t_k/x_k] \iff \exists x_1\ldots x_k(x_1 = t_1 \wedge \ldots \wedge x_k = t_k \wedge A),$$

provided that the variables $x_1,\ldots,x_k$ *do not occur in the terms* $t_1,\ldots,t_k$.

PROOF. This is a standard result of first-order logic which also holds for $L_{\omega_1\omega}$. □

We will not give completely formal proofs of theorems in $L_{\omega_1\omega}$, but will partly resort to informal arguments. We will, however, try to make these arguments correspond as closely as possible to formal constructions of proofs in $L_{\omega_1\omega}$. Because proofs in $L_{\omega_1\omega}$ may be infinitely long, a completely formal proof by exhibiting the sequence of formulas constituting the proof cannot usually even be given. Instead we have to use mathematical induction, by which the existence of a certain proof sequence can be shown.

The deduction theorem will be used in an informal way, by temporarily regarding the variables free in assumption formulas as constants. This means that we are not allowed to use the rule GN to universally quantify variables that occur free in assumption formulas.

CHAPTER 3

DESCRIBING STATE TRANSFORMATIONS

The language of *descriptions* will be defined in this chapter, the syntax in Section 3.1 and the semantics in Section 3.2. The language will be nondeterministic, mainly because we allow program specifications to occur as parts of descriptions and we do not want to require these to be deterministic.

The language will contain a new kind of primitive statement called an *atomic description*. It can roughly be described as a nondeterministic assignment statement with an associated change of scope. In addition to this, the language contains the usual control structures of composition, selection and iteration, together with nondeterministic binary choice.

The semantics of the descriptions will be of the denotational kind, making use of the approximation relation for nondeterministic state transformations defined in PLOTKIN [37]. We will be following DE BAKKER [12] quite closely, the main deviations resulting from the fact that we have to consider state transformations between different state spaces and that we do not require the nondeterminism to be bounded. The latter has a profound effect on the semantics, to be discussed in the last section of the chapter.

3.1. SYNTAX OF DESCRIPTIONS

We will first introduce some special terminology for finite sequences of elements, as we are going to need this kind of construction quite often in the subsequent analysis. A finite sequence of elements of a set A will be called a *list* of elements of A. If x is a list, then $\ell(x)$ is the length of the list, and the elements of the list x are $x_1,\ldots,x_{\ell(x)}$, in this order. We use angular brackets for lists, i.e. $x = \langle x_1,\ldots,x_{\ell(x)}\rangle$. The empty list, with $\ell(x) = 0$, is denoted <>. The set of elements in a list x is denoted $\tilde{x}$.

For any function f: A → B, the *extension* of f to a function from lists of elements of A to lists of elements of B is defined by

$$f(\langle x_1,\ldots,x_{\ell(x)}\rangle) = \langle f(x_1),\ldots,f(x_{\ell(x)})\rangle,$$

where $x_1,\ldots,x_{\ell(x)}$ are elements of A. If x and y are lists of elements of A, then

$$\langle x,y\rangle \equiv \langle x_1,\ldots,x_{\ell(x)},y_1,\ldots,y_{\ell(y)}\rangle,$$

and if $\ell(x) = \ell(y)$,

$$\langle x/y\rangle \equiv \langle x_1/y_1,\ldots,x_{\ell(x)}/y_{\ell(y)}\rangle$$

and

$$x = y \equiv x_1 = y_1 \wedge \ldots \wedge x_{\ell(x)} = y_{\ell(y)}.$$

Let from now on L be some fixed $L_{\omega_1\omega}$ language. If t is a term of L, then var(t) is the set of all variables occurring in t. Similarly, if Q is a formula of L, then var(Q) is the set of all variables free in Q.

The set of *descriptions* (in L) is defined by induction as follows:

(i) If x and y are lists of distinct variables, $\tilde{x} \cap \tilde{y} = \emptyset$, and Q is a first-order formula of L, then

$$x/y.Q \qquad \textit{(atomic description)}$$

is a description.

(ii) If S and S' are descriptions and B is a first-order formula of L, then

(S;S')	*(composition)*
(S ∨ S')	*(nondeterministic choice)*
(B → S | S')	*(selection)*
(B * S)	*(iteration)*

are descriptions.

We use descriptions to describe state transformations. A state is essentially a collection of variables, together with the values assigned to these variables. A state transformation may change the values assigned to the variables in the state, but it may also change the collection of variables in the

state, by adding some variables and removing some others. Programs will form a special kind of descriptions. However, descriptions are more general than programs, in that we can express almost any input/output relation as a description (including all relations definable by a first-order formula of L). This generality makes it possible to also express program specifications as descriptions.

The atomic description $x/y.Q$ is the source of this generality. The effect of this, applied to a state with the set of variables V, is roughly as follows. First, the new variables in the list x, i.e. those variables not already in the set V, are added to the state. Then new values are assigned to the variables x, the values being chosen so that the condition Q will become true. Finally, the variables y are removed from the state. The set of variables in the new state will thus be $(V \cup \tilde{x}) - \tilde{y}$. If there is more than one assignment of values to x which makes Q true, one of these is chosen nondeterministically. If there is no such assignment, the computation is considered not to terminate.

As an example, consider the atomic description

$$u,w/v.\,(0 \leq u+w \leq v+z)$$

applied to an initial state with variables $V = \{u,v,z\}$ (the list brackets are omitted in examples, i.e. we write u,w/v above instead of $\langle u,w\rangle/\langle v\rangle$). The new state will then have the variables $W = \{u,z,w\}$, i.e. the variable w has been added and v has been deleted. If initially we have that $(u,v,z) = (1,1,1)$ then in the new state we have $(u,z,w) = (a,1,b)$, where a and b are chosen so that the condition $0 \leq a + b \leq 2$ is satisfied. If the values are chosen from the set of natural numbers, then there are only three possible assignements of values to the variables (u,z,w): (0,1,2), (1,1,1) and (2,1,0).

The descriptions (S;S'), $(B \rightarrow S|S')$ and (B*S) provide the basic control structures of programming languages (we write $(B \rightarrow S|S')$ for if B then S else S' and (B*S) for while B do S). The description $(S \vee S')$ is a nondeterministic choice between executing S or S'.

Let S be a description and let V be a set of variables. The set of variables fin(S,V) is then defined as follows. First, if $V = \emptyset$ then $fin(S,V) = \emptyset$. For nonempty V we define fin(S,V) by cases as follows:

(1) $$fin(x/y.Q,V) = \begin{cases} (V-\tilde{y}) \cup \tilde{x}, & \text{if } var(Q) \subseteq V \cup \tilde{x},\ \tilde{y} \subseteq V \\ \emptyset & \text{otherwise} \end{cases}$$

(2) $$fin(S';S'',V) = fin(S'',fin(S',V))$$

(3) $$fin(S' \vee S'',V) = \begin{cases} fin(S',V), & \text{if } fin(S',V) = fin(S'',V) \\ \emptyset & \text{otherwise} \end{cases}$$

(4) $$fin(B \to S' \mid S'',V) = \begin{cases} fin(S',V), & \text{if } var(B) \subseteq V,\ fin(S',V) = fin(S'',V) \\ \emptyset & \text{otherwise} \end{cases}$$

(5) $$fin(B*S',V) = \begin{cases} V, & \text{if } fin(S',V) = V \text{ and } var(B) \subseteq V \\ \emptyset & \text{otherwise.} \end{cases}$$

Let V and W be two sets of variables, $V,W \neq \emptyset$. Then S is said to be a *legal* description from V to W, denoted $S: V \to W$, if $fin(S,V) = W$. The set V of variables is said to be a *legal initial space* for the description S, if $fin(S,V) \neq \emptyset$. The set W is said to be the *final space* of the description S for the initial space V, if $fin(S,V) = W$. Intuitively, $S: V \to W$ says that the initial states of the transition S contain the variables V and the final states of S contain the variables W.

If S is a legal description from V to W, then each component description of S will be assigned a unique initial legal space determined by S and V, and consequently also a unique final space. The initial and final spaces of the components of a description $S: V \to W$ are determined as follows:

(1) If $S = (S';S'')$, then $S': V \to fin(S',V)$ and $S'': fin(S',V) \to W$.

(2) If $S = (S' \vee S'')$, then $S': V \to W$ and $S'': V \to W$.

(3) If $S = (B \to S' \mid S'')$, then $S': V \to W$ and $S'': V \to W$.

(4) If $S = (B*S')$, then $S': V \to W$.

3.2. SEMANTICS OF DESCRIPTIONS

We start again by fixing our terminology and introducing some notations, this time for relations. Let D be a nonempty set, and let R be a relation in D, i.e. $R \subseteq D \times D$. Then R is said to be

reflexive, if $d\,R\,d$, for each $d \in D$,

transitive, if $d\,R\,d'$ and $d'\,R\,d''$ implies $d\,R\,d''$, for any $d,d',d'' \in D$,

symmetric, if $d\,R\,d'$ implies $d'\,R\,d$, for any $d,d' \in D$, and

antisymmetric, if $d\,R\,d'$ and $d'\,R\,d$ implies $d = d'$, for any $d,d' \in D$.

The relation R is a *preorder*, if it is reflexive and transitive. It is a *partial order*, if it is also antisymmetric. If it is a preorder, and in addition is symmetric, then it is an *equivalence relation*.

Let now V be a nonempty set of variables, and let D be some nonempty set. Then the *state space* determined by V and D, denoted V_D, is defined as $V_D = D^V \cup \{\perp_{V,D}\}$. Here D^V is as before the set of all V-assignments in D, i.e. the set of all functions $s: V \to D$, while $\perp_{V,D}$ is a special element not belonging to D^V, which is introduced for the purpose of modeling nontermination. The elements in D^V are called *proper states*, while $\perp_{V,D}$ is called the *undefined state*. The subscripts of the undefined state will be omitted when it is clear from the context to which state space the undefined state belongs.

For the purpose of the present section, we can think of $\perp$ as signalling the possiblity of nontermination. Thus, if A is the set of possible final states of a computation, we will add to A the undefined state if and only if there is a possibility that the computation may not terminate. In the next section this will be shown to be only approximately true, but for the present section this intuitive explanation could be helpful.

The set of all nonempty subsets of V_D will be denoted $P_D(V)$. Let W be a nonempty set of variables. A *(nondeterministic) state transformation* from V_D to W_D will be identified with a function $f: V_D \to P_D(W)$, satisfying the condition $f(\perp_{V,D}) = \{\perp_{W,D}\}$. For each proper state $s \in V_D$, f(s) will be the set of all possible final states of the state transformation. We denote the set of all state transformations from V_D to W_D with $F_D(V,W)$.

A *state predicate* on V_D is a function $f: V_D \to Tr$, satisfying the condition $f(\perp_{V,D}) = ff$. The set of all state predicates on V_D is denoted $E_D(V)$. Intuitively, a state predicate is an assertion about the values of the variables in the state.

The semantical definition of the descriptions will require some preliminary work, mainly necessiated by the iteration. We start by defining

some ways of constructing new state transformations from old ones. The fact that these constructions really are state transformations is easily verified.

The state transformations $\Omega_{V,D}$, $\Lambda_{V,D}$ in $F_D(V,V)$ are defined by

$$\Omega_{V,D}(s) = \{\perp_{V,D}\}, \quad \Lambda_{V,D}(s) = \{s\}, \quad \text{for each } s \in V_D.$$

If $f \in F_D(V,V')$ and $f' \in F_D(V',V'')$, then $f;f' \in F_D(V,V'')$ is defined by

$$(f;f')(s) = \bigcup_{s' \in f(s)} f'(s'), \quad \text{for each } s \in V_D.$$

If f and f' are elements in $F_D(V,W)$, then $f \vee f' \in F_D(V,W)$ is defined by

$$(f \vee f')(s) = f(s) \cup f'(s), \quad \text{for each } s \in V_D.$$

Finally, if $b \in E_D(V)$ and $f, f' \in F_D(V,W)$, then $(b \to f | f') \in F_D(V,W)$ is defined by

$$(b \to f | f')(s) = \begin{cases} f(s), & \text{if } b(s) = tt \\ f'(s), & \text{if } b(s) = ff. \end{cases}$$

Next, we define a relation of *approximation* in $P_D(V)$ and $F_D(V,W)$. If U and U' are elements of $P_D(V)$, the U is said to *approximate* U', denoted $U \sqsubseteq U'$, if either $\perp \in U$ and $U - \{\perp\} \subseteq U'$ or $\perp \notin U$ and $U = U'$.

If f and f' are elements of $F_D(V,W)$, then f is said to *approximate* f', denoted $f \sqsubseteq f'$, if $f(s) \sqsubseteq f'(s)$ for every $s \in V_D$.

LEMMA 3.1. *Approximation is a partial order in* $P_D(V)$ *and* $F_D(V,W)$.

To get an intuitive idea of this relation, consider a nondeterministic computation proceeding at a certain speed, where all alternatives are simultaneously computed (i.e. the computation branches at choice points). Consider two time intervals t and t', $t < t'$. Let U be the set of final states reached at t, and U' the corresponding set at t'. If in U (or U') there is an unfinished computation going on, then U (or U') is also to include the undefined state. If now $\perp \notin U$, then all computations have been finished at time t. Therefore the set of final states at t' must be the same as the set of final states at t, i.e. $U = U'$. If on the other hand $\perp \in U$, then any final

states reached at t must be a final state at t' too, although there might be other final states at t', created by the unfinished computations at t. Thus we have that $U - \{\bot\} \subseteq U'$. All in all, we have that $U \sqsubseteq U'$. In general, $U \sqsubseteq U'$ means that U' could be a later result set than U for some nondeterministic computation. (The approximation relation is treated in more details in e.g. DE BAKKER [12] or PLOTKIN [37].)

The least element in $P_D(V)$ is the element $\{\bot\}$ of $P_D(V)$. This follows from the fact that for any $U \in P_D(V)$, $\{\bot\} - \{\bot\} = \emptyset \subseteq U$, i.e. $\{\bot\} \sqsubseteq U$. As a consequence of this, $\Omega_{V,D}$ will be the least element of $F_D(V,V)$.

LEMMA 3.2. *If* $f \sqsubseteq f'$ *and* $g \sqsubseteq g'$, *then* $f;g \sqsubseteq f';g'$, *for any* f, f', g *and* g' *in* $F_D(V,V)$.

PROOF. Assume that $f \sqsubseteq f'$ and $g \sqsubseteq g'$. Consider first the case when $\bot \in f;g(s)$ for $s \in V_D$. Assume that $f;g(s) \neq \{\bot\}$ (otherwise we have directly that $f;g(s) \sqsubseteq f';g'(s)$), and let $s'' \in f;g(s)$, $s'' \neq \bot$. This means that for some $s' \in f(s)$, $s' \neq \bot$, $s'' \in g(s')$. Thus by assumption we have that $s' \in f'(s)$ and also that $s'' \in g'(s')$, i.e. $s'' \in f';g'(s)$. Therefore $f;g(s) \sqsubseteq f'g'(s)$.

On the other hand, if $\bot \notin f;g(s)$, then $\bot \notin f(s)$ and for any $s' \in f(s)$ we must have that $\bot \notin g(s')$. By the definition of f;g this then gives that $f;g(s) = f';g'(s)$. Therefore, we also have in this case that $f;g(s) \sqsubseteq f';g'(s)$. □

LEMMA 3.4. *If* $f \sqsubseteq f'$ *and* $g \sqsubseteq g'$, *then* $(b \rightarrow f|g) \sqsubseteq (b \rightarrow f'|g')$, *for any* f, f', g *and* g' *in* $F_D(V,V)$ *and* b *in* $E_D(V)$.

PROOF. The result follows directly by considering the two cases b(s) = tt and b(s) = ff. □

Let $U_i \in P_D(V)$ for $i < \omega$, such that $U_0 \sqsubseteq U_1 \sqsubseteq \ldots \sqsubseteq U_n \sqsubseteq \ldots$. We define

$$\bigsqcup_{n<\omega} U_n = \bigcup_{n<\omega} U_n,$$

if $\bot \in U_n$ for each $n < \omega$ and

$$\bigsqcup_{n<\omega} U_n = U_k$$

otherwise, where U_k is the first element in the sequence not containing $\bot$. Obviously $\bigsqcup_{n<\omega} U_n$ will be an element of $P_D(V)$.

For $f_i \in F_D(V,V)$, $i < \omega$, such that $f_0 \sqsubseteq f_1 \sqsubseteq \ldots \sqsubseteq f_n \sqsubseteq \ldots$, we define $\bigsqcup_{n<\omega} f_n$ in $F_D(V,V)$ by

$$(\bigsqcup_{n<\omega} f_n)(s) = \bigsqcup_{n<\omega} f_n(s) \quad \text{for each } s \in V_D.$$

(Actually $\bigsqcup f_n$ is the least upper bound of the chain $f_0 \sqsubseteq f_1 \sqsubseteq \ldots \sqsubseteq f_n \sqsubseteq \ldots$ and similarly for $\bigsqcup U_n$, but as this is not needed in the sequel, it will not be proved.)

Let $b \in E_D(V)$ and $f \in F_D(V,V)$. We define the transformations $(b*f)^n$ in $F_D(V,V)$, as follows:

$$(b*f)^0 = \Omega_{V,D}$$

and

$$(b*f)^{n+1} = (b \to f; (b*f)^n \mid \Lambda_{V,D}), \quad \text{for } n \geq 0.$$

We will prove that

$$(b*f)^n \sqsubseteq (b*f)^{n+1} \quad \text{for } n \geq 0.$$

First, because $\Omega_{V,D}$ is the least element of $F_D(V,V)$, we have that

$$(b*f)^0 \sqsubseteq (b*f)^1.$$

Assuming that $(b*f)^n \sqsubseteq (b*f)^{n+1}$, $n \geq 0$, we have by Lemma 3.2 that

$$f;(b*f)^n \sqsubseteq f;(b*f)^{n+1},$$

from which we get by Lemma 3.3 that

$$(b \to f;(b*f)^n \mid \Lambda_{V,D}) \sqsubseteq (b \to f;(b*f)^{n+1} \mid \Lambda_{V,D}),$$

i.e. we have

$$(b*f)^{n+1} \sqsubseteq (b*f)^{n+2}.$$

The required result then follows by induction.

The state transformation (b*f) in $F_D(V,V)$, where $b \in E_D(V)$ and $f \in F_D(V,V)$, can now be defined by

$$(b*f) = \bigsqcup_{n<\omega} (b*f)^n.$$

Let now $M = \langle D,I \rangle$ be a structure for L. A formula Q with $var(Q) \subseteq V$, V a nonempty set of variables, can be interpreted as a state predicate in $E_D(V)$, denoted $int_M(Q,V)$, as follows:

$$int_M(Q,V)(s) = val_M(Q,s), \quad \text{for each } s \in V_D,\ s \neq \perp.$$

(For $s = \perp$ we always have $int_M(Q,V)(s) = ff$, by the definition of state predicates.)

Let $x/y.Q:V \to W$ be an atomic description, $x = \langle x_1,\ldots,x_k \rangle$, and let s be a state in V_D. The effect of the atomic description is to compute some new state s' in W_D, where $s'(x_1),\ldots,s'(x_k)$ are chosen so that the condition Q will be satisfied, while $s'(z) = s(z)$ for variables z in W not occurring in the list x. More precisely, s' is said to be a *possible choice* of the atomic description, for initial state s in V_D, if

$$val_M(Q,s\langle s'(x)/x \rangle) = tt$$

and

$$s'(z) = s(z), \quad \text{for every } z \in W - \tilde{x}$$

A legal description $S: V \to W$, V and W nonempty sets of variables, will be interpreted as a state transformation in $F_D(V,W)$, denoted $int_M(S,V)$. We define the interpretation by cases as follows:

(i) $$int_M(x/y.Q,V)(s) = \begin{cases} W(s), & \text{if } W(s) \neq \emptyset \\ \{\perp\}, & \text{if } W(s) = \emptyset, \end{cases}$$

where $W(s) \subseteq W_D$ is the set of all possible choices of x/y.Q for initial state $s \in V_D$, $s \neq \perp$.

(ii) $$int_M(S';S'',V) = int_M(S',V);\ int_M(S'',fin(S',V)),$$
$$int_M(S' \vee S'',V) = int_M(S',V) \vee int_M(S'',V),$$
$$int_M(B \to S' \mid S'',V) = (int_M(B,V) \to int_M(S',V) \mid int_M(S'',V)),$$

$$int_M(B*S',V) = (int_M(B,V) * int_M(S',V)).$$

3.3. STRONG AND WEAK TERMINATION

As explained in the previous section, the undefined state $\perp$ is used to indicate the possibility of nontermination. That is, for any description $S: V \to W$ and any initial state $s \in V_D$, $\perp \in int_M(S,V)(s)$ if and only if there is an execution of S from initial state s which does not terminate. On closer inspection, however, it turns out that this is not really the case. More precisely, for some descriptions $S: V \to W$ and some initial states $s \in V_D$ we have that $\perp \in int_M(S,V)(s)$, although execution of S from initial state s seems to be guaranteed to terminate.

As an example, consider the following program (discussed in DIJKSTRA [15, p. 77]):

```
S': while x ≠ 0 do
       if x ≥ 0 then x := x-1
                else x := "any non-negative integer" fi od.
```

The variable x is assumed to range over the set of integers. This program should obviously terminate for any initial value of x, be it positive, zero or negative.

The program S' can be written as a description $S: V \to V$, where $V = \{x\}$:

```
S: (x ≠ 0 * S1)
```

where

```
S1: (x ≥ 0 → S2|S3),
S2: x'/<>.(x' = x-1); x/x'.(x = x')
S3: x/<>.(x ≥ 0).
```

The description S2 corresponds to the assignment x := x-1 and S3 to the assignment x := "any non-negative integer".

Choose the structure $M = \langle D,I \rangle$ to be the standard one, i.e. D is the set of integers and I assigns the usual interpretation to the operations and relations occurring in S, S1, S2 and S3. Let us for simplicity identify a state $s: \{x\} \to D$ with $s(x)$, so that $V_D = D \cup \{\perp\}$. We will now compute

$int_M(S,V)(-1)$.

It is straightforward to verify that

$$int_M(S1,V)(a) = \begin{cases} \{a-1\}, & \text{if } a \geq 0 \\ \{0,1,2,\ldots\}, & \text{if } a < 0. \end{cases}$$

Let now $f = int_M(S,V)$ and let f^i be the successive approximates of f, i = 0,1,2,..., as described in the previous section. We then have that

$$f^0(-1) = \{\perp\},$$

$$f^1(-1) = \{\perp\},$$

$$f^2(-1) = \{0,\perp\},$$

$$\vdots$$

$$f^i(-1) = \{0,\perp\},$$

$$\vdots$$

Thus we get that

$$f(-1) = \bigsqcup_{i=0}^{\infty} f^i(-1) = \{0,\perp\}.$$

The result set thus contains $\perp$. This contradicts out intuition about the behaviour of S', which we expect to be guaranteed to terminate for the initial state x = -1.

A closer look at the semantics of the iteration statement, as it was defined in the previous section, shows that the notion of termination captured by the definition is more restrictive than the usual notion of termination. The notion of termination in the definition is called strong termination and is characterized as follows: A loop is said to *terminate strongly* if for any initial state s there is an integer N_s such that the loop for this initial state is guaranteed to terminate in less than N_s iterations (DIJKSTRA [16]). The semantics given for the iteration statement consequently identifies nontermination with termination which is not strong *(weak termination)*;

the result set contains $\perp$ if and only if termination of the iteration statement is not strong. Weak termination in the example program S' above is due to the statement x := "any non-negative integer". Because of this statement, no upper bound can be given for the number of iterations required for termination in initial state x = -1.

The problem of weak termination can be avoided, if we restrict ourselves to state transformations of *bounded nondeterminacy*, i.e. state transformations $f \in F_D(V,W)$ satisfying the condition that either f(s) is finite or $\perp \in f(s)$, for any $s \in V_D$. In this case one can safely identify $\perp$ with nontermination, because weak termination is then not possible. This restriction is made by PLOTKIN [37] and also by DE BAKKER [12].

We could also choose this approach. The simplest way for us to achieve bounded nondeterminacy would be to restrict the basic statements x/y.Q in such a way that for any state there would only be a finite number of possible choices. We will say that the basic description is *finite* in the structure M if this is the case.

However, this approach is too restrictive for the purposes we have in mind, so we will continue to work with the general state transformations. We will also not change the semantics of descriptions given in the previous section, but accept the fact that it describes strong termination of loops rather than the usual notion of termination. It is possible to give a denotational semantics for descriptions in which the right notion of termination is captured (see BACK [4]), but we will not use this semantics here. There are essentially two reasons. First, it turns out that weak termination cannot be handled in the logic $L_{\omega_1\omega}$ [3], but would require an essentially stronger logic, a step we do not want to take. Secondly, it is possible to define a useful sublanguage of descriptions, in which the problems with weak terminations are avoided, without having to restrict all basic descriptions to be finite. We will return to these questions in Chapters 5 and 6.

CHAPTER 4

REFINEMENT AND WEAKEST PRECONDITIONS

In this chapter we will show that the correctness of a refinement step can be expressed as a binary relation of *refinement* between descriptions. This relation is based on a corresponding relation of refinement between state transformations. Total correctness of programs can be expressed using the refinement relation, as well as strong equivalence of programs.

Section 4.1 will be devoted to an explication of the notion of a correct refinement step. It will be shown that the refinement relation captures the intuitive idea of a refinement step being correct. The refinement relation will be a preorder, justifying the stepwise manner of program construction, as explained in the introduction.

In Section 4.2 the weakest precondition of a state transformation is defined. It is shown that refinement between state transformations can be characterized using weakest preconditions. This is a fundamental result, which will be used in the next chapter to give a general proof rule for refinement between descriptions.

4.1. REFINEMENT BETWEEN DESCRIPTIONS

A refinement step is considered to be correct, if it preserves the correctness of the program being refined. This informal notion can be made more precise as follows. Let Prog be the set of programs under consideration, let Spec be the set of program specifications and let Spec $\times$ Prog be a relation of satisfaction, i.e. R *sat* S holds if specification R is satisfied by program S. *Correct refinement* is then defined as the relation *ref* between programs S and S' in Prog,

$$S \; ref \; S' \text{ iff } \forall R \in \text{Spec } (R \; sat \; S \Rightarrow R \; sat \; S').$$

In words, S is *refined by* S' if and only if S' satisfies every specification that S satisfies.

LEMMA 4.1. *Correct refinement is a preorder in* Prog.

PROOF. Immediate consequence of the definition. □

This refinement relation is studied in BACK [2] for different choices of Prog, Spec and *sat*. Here we will only be interested in one specific refinement relation, the refinement relation which preserves *total correctness* of *nondeterministic programs*. We want to define correct refinement as a semantic notion, and will therefore choose Prog to be the set of state transformations $F_D(V,W)$. The relation will thus only be defined between state transformations with the same initial and final space.

To specify a nondeterministic program with respect to total correctness, we have to state the conditions under which the program is required to terminate strongly, and we have to specify the conditions which must hold upon termination of the program. For state transformations in $F_D(V,W)$, this means that we have to give a set $U \subseteq V_D$ for which f may not yield $\perp$, ans we have to specify for each $s \in U$ the set W_s of proper final states allowed, $W_s \subseteq W_D - \{\perp\}$. This information can be given in the form of a state transformation e in $F_D(V,W)$, where $U = \{s \in V_D \mid \perp \notin e(s)\}$ and $W_s = e(s)$. We can therefore also use $F_D(V,W)$ for Spec. The relation of satisfaction holds between e and f in $F_D(V,W)$ if $f(s) \subseteq e(s)$ for any $s \in U$. Equivalently, e *sat* f iff for any $s \in V_D$, $\perp \notin e(s) \Rightarrow f(s) \subseteq e(s)$.

It is easy to verify that for this choice of Prog, Spec and *sat*, the refinement relation actually coincides with the satisfaction relation. In other words, we have that for e and f in $F_D(V,W)$,

e *ref* f if and only if e *sat* f.

To see this, assume first that e *ref* f holds. This means that for any $h \in F_D(V,W)$, h *sat* e $\Rightarrow$ h *sat* f. If we choose h to be e, then we have that e *sat* e $\Rightarrow$ e *sat* f. As e *sat* e always holds, this gives us that e *sat* f. Conversely assume that e *sat* f holds. Let $U = \{s \in V_D \mid \perp \notin e(s)\}$. Let $h \in F_D(V,W)$ be such that h *sat* e, and let s be an element in U. Then we have by the assumptions that $e(s) \subseteq h(s)$ and $f(s) \subseteq e(s)$, i.e. $f(s) \subseteq h(s)$. Thus h *sat* f and conseuqently e *ref* f. This motivates the following definition.

DEFINITION 4.1. Let f and f' be state transformations in $F_D(V,W)$. Then f is *refined by* f', denoted $f \leq f'$, if for any $s \in V_D$,

$$\perp \notin f(s) \Rightarrow f'(s) \subseteq f(s).$$

DEFINITION 4.2. Let S and S' be two legal descriptions from V to W, and let M be a structure for L. We say that S is *refined* by S' in M, denoted $S \leq_M S'$, if $int_M(S,V) \leq int_M(S',V)$. We say that $S \leq S'$ is a *semantic consequence* of the set Δ of sentences, denoted $\Delta \models S \leq S'$, if $S \leq_M S'$ for any model M of Δ.

Thus $S \leq S'$ holds if whenever S terminates strongly for an initial state, S' also terminates strongly for this state, and any final state of S' for such an initial state is a possible final state of S too for this initial state (more concisely and less precisely, we could say that S' is *more defined* and *more deterministic* than S).

Any program specification given in the form of an entry condition and an exit condition can be expressed as a description (the way in which this is done is explained in Section 6.2). In fact, the main purpose of the atomic description is to make this possible. This means that total correctness of descriptions will be a special case of refinement between descriptions, i.e. $S \leq S'$ says that S' is totally correct with respect to S, when S is a description that expresses a program specification.

The refinement relation induces an equivalence relation in the obvious way. We say that the state transformations f and f' are *equivalent*, denoted $f \approx f'$, if $f \leq f'$ and $f' \leq f$ holds. Similarly, the descriptions S and S' are *equivalent in* M, denoted $S \approx_M S'$, if $S \leq_M S'$ and $S' \leq_M S$ holds. Finally, $S \approx S'$ is said to be a *semantic consequence* of Δ, denoted $\Delta \models S \approx S'$, if $\Delta \models S \leq S'$ and $\Delta \models S' \leq S$ holds.

If S and S' are equivalent, then S and S' will be guaranteed to terminate strongly for the same set of initial states, and have the same set of possible final states for any of these initial states. S and S' may differ, however, for initial states for which they are not guaranteed to terminate strongly.

For deterministic programs, $S \leq S'$ reduces to the usual approximation relation between deterministic state transformations, i.e. for any initial state for which S terminates, S' will also terminate and gives the same final state as S. $S \approx S'$ again reduces to strong equivalence between programs (see e.g. MANNA [34]), i.e. S and S' will terminate for the same

initial states and will give the same final states for these initial states.

In SMYTH [41] a relation similar to the refinement relation above is defined between state transformations of bounded nondeterminacy. Smyth uses it to prove the existence of a certain power domain construction under weaker assumptions than those made by PLOTKIN [37]. (The refinement relation here has been arrived at independently of the work by Smyth, and is also used for an entirely different purpose.)

4.2. WEAKEST PRECONDITIONS

Let f be a state transformation in $F_D(V,W)$ and let q be a predicate in $E_D(W)$. We define a predicate wp(f,q) in $E_D(V)$, called the *weakest precondition* of f for q, as follows: For any $s \in V_D$, $s \neq \perp$,

$$wp(f,q)(s) = tt \text{ iff for any } s' \in f(s),\ q(s') = tt.$$

As an immediate consequence of this definition, we see that if $wp(f,q)(s) = tt$ for $s \in V_D$, then $\perp \notin f(s)$, because $q(\perp) = ff$, by the definition of state predicates. This formulation of weakest preconditions for state transformations and state predicates is essentially the one given in DE BAKKER [12].

Let S be a description interpreted as a state transformation f in $F_D(V,W)$ and let Q be a condition interpreted as a predicate q in $E_D(V)$. Then wp(f,q) will be the set of all initial states in which S is guaranteed to terminate strongly, in a final state satisfying condition Q.

The boolean operations on truth values can be extended to state predicates in the obvious way: If p and p' are two state predicates in $E_D(V)$, then $(p \wedge p')$ is a state predicate in $E_D(V)$, defined by

$$(p \wedge p')(s) = p(s) \wedge p'(s), \quad \text{for each } s \in V_D,\ s \neq \perp.$$

Similarly for the other boolean connectives. It will be convenient to use the predicate p also as expressing the condition that $p(s) = tt$ for each $s \in V_D$. This is done below and will also be used later.

The following theorem shows that refinement can be characterized using weakest preconditions. This fact will be used in the next chapter to give a proof technique for the correctness of refinement steps.

THEOREM 4.3. *Let* f *and* f' *be state transformations in* $F_D(V,W)$. *Then* $f \leq f'$ *iff for any* q *in* $E_D(W)$,

$$wp(f,q) \Rightarrow wp(f',q).$$

PROOF.

($\Rightarrow$) Assume that $f \leq f'$ and let q be a predicate in $E_D(W)$. Let $s \in V_D$ be such that $wp(f,q)(s) = tt$. This means that $\perp \notin f(s)$, and using the assumption, this means that $f'(s) \subseteq f(s)$. Let now $s' \in f'(s)$. Then $s' \in f(s)$, and as $wp(f,q)(s) = tt$, we must have that $q(s') = tt$. Thus we have that $wp(f',q)(s) = tt$.

($\Leftarrow$) Assume that $wp(f,q) \Rightarrow wp(f',q)$ holds for any q in $E_D(W)$. Let $s \in V_D$ be such that $\perp \notin f(s)$. Define a state predicate q_s in $E_D(W)$ by $q_s(s') = tt$ iff $s' \in f(s)$, for any $s' \in W_D$, $s' \neq \perp$. This means that $wp(f,q_s)(s) = tt$, and by assumption, that $wp(f',q_s)(s) = tt$. Thus, for any $s' \in f'(s)$, $q_s(s') = tt$, i.e. for any $s' \in f'(s)$, we have that $s' \in f(s)$, which means that $f'(s) \subseteq f(s)$. This shows that $f \leq f'$. □

CHAPTER 5

PROVING REFINEMENT BETWEEN DESCRIPTIONS

The general proof rule for refinement between descriptions will be derived in this chapter. In Section 5.1 we will give rules for computing the weakest preconditions of descriptions, and show that these rules are correct. In Section 5.2 the proof rule for refinement is derived, and a soundness and completeness result is proved. The proof rule is based on the use of weakest preconditions of descriptions. We will also give a proof rule for equivalence of descriptions and present a useful induction rule for iteration, together with some other properties of refinement. In Section 5.3 the properties of weakest preconditions given by DIJKSTRA [15] are discussed. In Section 5.4 we finally prove an important replacement property of descriptions, which will provide a justification for the top-down program development strategy.

5.1. WEAKEST PRECONDITIONS OF DESCRIPTIONS

Let $S: V \to W$ be a description and Q a formula of L, $var(Q) \subseteq W$. Let M be a structure for L. The description S will then be interpreted as a state transformation $f = int_M(S,V) \in F_D(V,W)$, and the formula Q as a state predicate $q = int_M(Q,W) \in E_D(W)$. We may now ask for a formula P of L, $var(P) \subseteq V$, which describes the weakest precondition of f for q, i.e. we require that

$$int_M(P,V) = wp(f,q). \tag{5.1}$$

This formula P will then give the weakest precondition that an initial state must satisfy so that the execution of S is guaranteed to terminate strongly in a final state that satisfies condition Q. This section will be concerned with showing how such a condition P can be computed for any S and Q, and that the condition P computed satisfies (5.1).

We introduce the abbreviations true and false for sentences of $L_{\omega_1\omega}$ by

$$\text{true} =_{df} \forall v_0(v_0=v_0)$$

and

$$\text{false} =_{df} \sim \forall v_0(v_0=v_0).$$

Thus, true will hold for any proper state in any state space V_D, while false will hold for no state in any state space V_D.

Next we introduce the abbreviations skip and abort for descriptions, by

$$\text{skip} =_{df} <>/<>.\text{true}$$

and

$$\text{abort} =_{df} <>/<>.\text{false}.$$

Evidently, skip will be the identity transformation in $F_D(V,V)$ for any V, i.e.

$$\text{int}_M(\text{skip},V) = \Lambda_{V,D}$$

while abort will be the undefined state transformation in $F_D(V,V)$, i.e.

$$\text{int}_M(\text{abort},V) = \Omega_{V,D}.$$

Let B be a formula of L, $\text{var}(B) \subseteq V$, and let S be a description from V to V. Then the descriptions $(B*S)^n$ from V to V, $n < \omega$, are defined by

$$(B*S)^0 = \text{abort},$$

$$(B*S)^n = (B \rightarrow S;(B*S)^{n-1} \mid \text{skip}), \quad n > 0.$$

Using induction on n, it is easily verified that

$$\text{int}_M((B*S)^n,V) = (\text{int}_M(B,V)*\text{int}_M(S,V))^n, \quad \text{for } n < \omega.$$

DEFINITION 5.1. Let S be a legal description from V to W, $V,W \neq \emptyset$, and let R be a formula of L, $var(R) \subseteq W$. Then the *weakest precondition* of S for R, denoted WP(S,R), is defined by induction on the structure of S, as follows:

(i) $\quad WP(x/y.Q,R) = \exists xQ \wedge \forall x(Q \Rightarrow R)$,

(ii) $\quad WP(S';S'',R) = WP(S',WP(S'',R))$,

(iii) $\quad WP(S' \vee S'',R) = WP(S',R) \wedge WP(S'',R)$,

(iv) $\quad WP(B \rightarrow S' | S'',R) = (B \Rightarrow WP(S',R)) \wedge (\sim B \Rightarrow WP(S'',R))$,

(v) $\quad WP(B*S',R) = \bigvee_{n<\omega} WP((B*S')^n,R)$.

We make the convention that $\exists xQ = Q$ and $\forall x(Q \Rightarrow R) = (Q \Rightarrow R)$ in (i), when $x = <>$. Using this convention, we get from (i) that

$$WP(skip,R) = true \wedge (true \Rightarrow R) \iff R,$$

and

$$WP(abort,R) = false \wedge (false \Rightarrow R) \iff false,$$

for any formula R of L.

LEMMA 5.1. *If S is a legal description from V to W, $V,W \neq \emptyset$, and R is a formula of L, $var(R) \subseteq W$, then $WP(S,R)$ is a formula of L, with $var(WP(S,R)) \subseteq V$.*

PROOF. The proof goes by induction on the structure of S. We show here only the basis step, i.e. the case when $S = x/y.Q$. Because $var(Q) \subseteq V \cup \tilde{x}$, we have $var(\exists xQ) \subseteq V$, as no variable in x is free in $\exists xQ$. Also, because $var(R) \subseteq W$, and $W = (V-\tilde{y}) \cup \tilde{x} \subseteq V \cup \tilde{x}$, we have $var(\forall x(Q \Rightarrow R)) \subseteq var(Q) \cup var(R) - \tilde{x}$ i.e. $var(\forall x(Q \Rightarrow R)) \subseteq V$. This means that $var(WP(x/y.Q,R) \subseteq V$. The induction step, i.e. case (ii)-(v) in definition 5.1, is proved straightforwardly. □

We are now ready for the main result of this section, i.e. that condition (5.1) is satisfied by choosing WP(S,R) for P.

THEOREM 5.2. *Let S be a legal description from V to W, $V,W \neq \emptyset$, and let R be a formula of L, $var(R) \subseteq W$. Then, for any structure M of L, we have that*

$$int_M(WP(S,R),V) = wp(int_M(S,V),int_M(R,W)).$$

PROOF. The proof will go by induction on the structure of S. Let M = <D,I> be a structure for L. Let

$$f = int_M(S,V) \in F_D(V,W)$$

and

$$r = int_M(R,W) \in E_D(W).$$

We have to prove that

$$int_M(WP(S,R),V) = wp(f,r).$$

(i) S is x/y.Q. In this case we have that f(s) = W(s), if W(s) $\neq \emptyset$, and f(s) = {$\perp$}, if W(s) = $\emptyset$, where W(s) is the set of choices of x/y.Q for s, i.e.

$$s' \in W(s) \text{ iff } val_M(Q,s\langle s'(x)/x\rangle) = tt$$

and

$$s(z) = s'(z) \quad \text{for each } z \in W - \tilde{x}.$$

($\Rightarrow$) Let $s \in V_D$ such that $int_M(WP(S,R),V)(s) = tt$. This means that

$$val_M(\exists x Q,s) = tt \quad \text{and} \quad val_M(\forall x(Q \Rightarrow R),s) = tt,$$

using the definition of WP for the atomic description, and the definition of the interpretation of formulas.

Now $val_M(\exists xQ,s) = tt$ iff $val_M(Q,s\langle d/x\rangle) = tt$ for some list d of elements in D. If we choose $s' \in W_D$ by $s'(x_i) = d_i$, for $i = 1,\ldots,\ell(x)$, and $s'(z) = s(z)$ for $z \in W - \tilde{x}$, we have that $val_M(Q,s\langle s'(x)/x\rangle) = tt$, i.e. $s' \in W(s)$. Therefore $W(s) \neq \emptyset$, and we have that $f(s) = W(s)$.

Assume now that $s' \in W(s)$, which implies that $s \neq \perp$. Then $s'(z) = s(z)$ for $z \in W - \tilde{x}$, and $val_M(Q,s\langle s'(x)/x\rangle) = tt$. By assumption,

$val_M(\forall x(Q \Rightarrow R),s) = tt$, i.e. $val_M(Q \Rightarrow R, s\langle d/x\rangle) = tt$ for any list d of elements in D. This means that $val_M(R, s\langle s'(x)/x\rangle) = tt$, by choosing $d = s'(x)$ and using modus ponens. Because $s\langle s'(x)/x\rangle(z) = s'(z)$ for any $z \in W$, and $var(R) \subseteq W$, this means that $val_M(R,s') = tt$, i.e. $int_M(R,W)(s') = tt$. Thus we have $r(s') = tt$ and $wp(f,r)(s) = tt$, as s' was an arbitrarily chosen element of f(s).

($\Leftarrow$) Let $s \in V_D$ such that $wp(f,r)(s) = tt$. This means that for any $s' \in f(s)$, $r(s') = tt$. Therefore we have that $\perp \notin f(s)$, because $r(\perp) = ff$. Thus $W(s) \neq \emptyset$, i.e. there is an $s' \in W_D$ such that $val_M(Q, s\langle s'(x)/x\rangle) = tt$ and $s'(z) = s(z)$ for $z \in W - \tilde{x}$. Thus $val_M(\exists x Q, s) = tt$.

Assume that $val_M(Q, s\langle d/x\rangle) = tt$. Define $s' \in W_D$ by $s'(z) = s(z)$ for $z \in W - \tilde{x}$, and $s'(x_i) = d_i$ for $i = 1,\ldots,\ell(x)$. Then $s' \in f(s)$, which implies that $r(s') = tt$, i.e. $val_M(R,s') = tt$. Because $var(R) \subseteq W$ and $s\langle d/x\rangle(z) = s'(z)$ for $z \in W$, we have from this that $val_M(R, s\langle d/x\rangle) = tt$. This gives $val_M(Q \Rightarrow R, s\langle d/x\rangle) = tt$, i.e. we have that $val_M(\forall x(Q \Rightarrow R),s) = tt$, as d was arbitrarily chosen. Thus we have proved that $int_M(WP(S,R),V)(s) = tt$.

(ii) S is S';S", where S': $V \to V'$ and S": $V' \to W$. Define $f' = int_M(S',V)$ and $f'' = int_M(S'',V')$. Then

$$int_M(WP(S';S'',R),V) = int_M(WP(S',WP(S'',R)),V)$$

$$= wp(f', int_M(WP(S'',R),V')) \quad \text{(by induction hyp.)}$$

$$= wp(f', wp(f'',r)) \quad \text{(by induction hyp. again)}.$$

Let $s \in V_D$. We then have that $wp(f',wp(f'',r))(s) = tt$ iff for each $s' \in f'(s)$, $wp(f'',r)(s') = tt$, iff for each $s' \in f'(s)$, $s'' \in f''(s')$, $r(s'') = tt$, iff for each $s'' \in f';f''(s)$, $r(s'') = tt$, iff $wp(f';f'',r)(s) = tt$. Thus we get that $wp(f',wp(f'',r)(s) = wp(f';f'',r)(s)$, i.e. $int_M(WP(S,R),V) = wp(f,r)$.

(iii) S is (B*S'). Let $int_M(S',V) = f'$ and $int_M(B,V) = b$. We first prove that for each $n < \omega$,

$$(5.2) \qquad int_M(WP((B*S')^n,R),V) = wp((b*f')^n,r),$$

by induction on n.

For n = 0 we have $(B*S')^0$ = abort. Let $s \in V_D$. We have that

$$int_M(WP((B*S')^0,R),V)(s) = int_M(false,V)(s) = ff.$$

On the other hand, we have

$$wp((b*f')^0,r)(s) = wp(\Omega_{V,D},r)(s) = ff.$$

Thus, for n = 0 we have that (5.2) holds.

Assume that (5.2) holds for $n \geq 0$. Then

$$int_M(WP((B*S')^{n+1},R),V)$$

$$= int_M(WP((B \to S';(B*S')^n | skip),R),V)$$

$$= int_M((B \Rightarrow WP(S';(B*S')^n,R)) \wedge (\sim B \Rightarrow R),V)$$

$$= (b \Rightarrow wp(f',int_M(WP((B*S')^n,R),V)) \wedge (\sim b \Rightarrow r)$$

$$= (b \Rightarrow wp(f',wp((b*f')^n,r))) \wedge (\sim b \Rightarrow r) \quad \text{(induction hyp.)}$$

$$= wp((b \Rightarrow f';(b*f')^n | \Lambda_{V,D}),r)$$

$$= wp((b*f')^{n+1},r).$$

Thus (5.2) holds for any $n < \omega$.

To prove this case, we have to show that $int_M(WP(B*S',R),V) = wp(b*f',r)$, where

$$WP(B*S',R) = \bigvee_{n<\omega} WP((B*S')^n,R).$$

First let $s \in V_D$ be such that $int_M(WP(B*S',R),V)(s) = tt$. This means that $int_M(WP((B*S')^n,R),V)(s) = tt$ for some $n \geq 0$. Therefore, by the previous result, we must have that $wp((b*f')^n,r)(s) = tt$. More particularly, this means that $\perp \notin (b*f')^n(s)$ and thus that

$$(b*f')(s) = (b*f')^n(s),$$

by the definition of (b*f'). Thus we get that wp(b*f',r)(s) = tt.

On the other hand, assume that $s \in V_D$ is such that $int_M(WP(B*S',R),V)(s) = ff$. This means that $int_M(WP((B*S')^n,R),V)(s) = ff$ for every $n < \omega$, i.e. $wp((b*f')^n,r)(s) = ff$ for every $n < \omega$. Assume first that $\perp \in (b*f')^n(s)$ for every $n < \omega$. In this case we have that

$$(b*f')(s) = \bigcup_{n<\omega} (b*f')^n(s),$$

and thus $\perp \in (b*f')(s)$. Therefore $wp(b*f',r)(s) = ff$. If, on the other hand, $\perp \notin (b*f')^n(s)$ for some n, then we have

$$(b*f')(s) = (b*f')^n(s).$$

In this case again, we have that $wp(b*f',r)(s) = wp((b*f')^n,r)(s) = ff$.

Thus we conclude that $int_M(WP(B*S',R),V)(s) = wp(b*f',r)(s)$ for each $s \in V_D$, which proves this case.

The proofs of the remaining two cases, (S'∨S") and (B → S|S"), do not present any greater difficulties and are therefore omitted. □

A similar theorem is proved in DE BAKKER [12]. However, the situation considered here is sufficiently different from the one considered by de Bakker to motivate a new proof of this central theorem. De Bakker proves the result for a programming language with assignment statement and recursion, and uses a model in which bounded nondeterminacy is assumed, whereas our language contains the atomic description and only a simple loop, and we do not assume bounded nondeterminacy. Also, by using an infinitary logic, we get a more natural expression of the weakest preconditions for loops.

Theorem 5.2 shows that the weakest preconditions of descriptions are expressible in the logic $L_{\omega_1\omega}$. The definition of weakest preconditions used here requires strong termination of a description. A natural question is whether it would be possible to express the weakest precondition with respect to general termination (strong or weak) in $L_{\omega_1\omega}$. However, as shown in BACK [3], this is not possible. To get expressibility, one has to use an essentially stronger logic, such as the logic $L_{\omega_1\omega_1}$, in which quantification over infinite sequences of variables is allowed. Another possibility, using disjunction over the class of all ordinals, is described by BOOM [8]. In either case, one looses the advantages of working in $L_{\omega_1\omega}$, i.e. completeness of the logic and the simple characterization of the weakest precondition of

loops. This is the most important reason why we choose to work with strong termination rather than with termination in general.

The use of $L_{\omega_1\omega}$ in connection with program correctness has been pioneered by ENGELER [17,18]. His approach has been further developed by SALWICKI [39] and the group in Warschau working on *algorithmic logic* [5]. In these approaches a new programming logic is designed, in which infinitary proof rules are used to handle total correctness of loops. The approach we take is different, in that we stay in the logic $L_{\omega_1\omega}$, translating total correctness assertions into formulas of this logic, rather than inventing a special logic for these correctness assertions. Weakest preconditions are studied in more detail by DE BAKKER [12], HOARE [26] and HAREL [21], the last mentioned in the context of *dynamic logic* [38].

5.2. A PROOF RULE FOR REFINEMENT

Let S and S' be legal descriptions from V to W, where V and W are assumed to be nonempty *finite* sets of variables. Let M = <D,I> be a structure for L. By definition 4.2, we have that $S \leq_M S'$ iff

$$int_M(S,V) \leq int_M(S',V).$$

By Theorem 4.3 we have that this again holds iff

(5.3) $\quad wp(int_M(S,V),q) \Rightarrow wp(int_M(S',V),q), \quad$ for any $q \in E_D(W)$.

Let G be a new k-place predicate symbol, where k is the number of variables in W, and let w be a list of distinct variables such that $\tilde{w} = W$. Let L' be the expansion of L that we get by adding G to the nonlogical symbols of L. Then G(w) is a formula of L'. For any choice of $q \in E_D(W)$, we can define an expansion M' of M to L', such that $int_{M'}(G(w),W) = q$. We achieve this by defining $I'(G)(a_1,\ldots,a_k) = tt$ iff $q(s) = tt$, where $s(w_i) = a_i$, for $i = 1,\ldots,k$. Then for any proper $s \in W_D$, $int_{M'}(G(w),W)(s) = val_{M'}(G(w),s) = I'(G)(s(w_1),\ldots,s(w_k)) = q(s)$. Conversely, in any expansion M' of M to L', the interpretation in M' of G(w) will be some predicate in $E_D(W)$. Therefore we have that (5.3) is equivalent to

$$wp(int_{M'}(S,V),int_{M'}(G(w),W)) \Rightarrow wp(int_{M'}(S',V),int_{M'}(G(w),W))$$

for any expansion M' of M to L'.

We have here used the fact that $int_{M'}(S,V) = int_{M}(S,V)$ and the same for S', as G is a new symbol that cannot occur in S or S'.

Using Theorem 5.2, we finally get that (5.3) is equivalent to

$$int_{M'}(WP(S,G(w)),V) \Rightarrow int_{M'}(WP(S',G(w)),V),$$

for any expansion M' of M to L'.

We formulate this result as a theorem.

THEOREM 5.3. *Let* S *and* S' *be legal descriptions from* V *to* W, *where* V *and* W *are finite nonempty sets of variables. Let* L' *be an expansion of* L *that we get by adding a new* k*-place predicate symbol* G *to the nonlogical symbols of* L, *where* k *is the number of variables in* W. *Let* w *be a list of distinct variables, such that* $\tilde{w} = W$. *Then* $S \leq_M S'$ *iff*

$$WP(S,G(w)) \Rightarrow WP(S',G(w))$$

holds in any expansion M' *of* M *to* L'.

Now, let Δ be a set of sentences of L. Then $\Delta \models S \leq S'$ iff

$S \leq_M S'$ for any model M of Δ.

This is, by Theorem 5.3, the case iff

(5.4) the assertion $WP(S,G(w)) \Rightarrow WP(S',G(w))$ holds in any expansion M' of M to L, for any model M of Δ.

Because Δ is a set of sentences of L, we have that if M' is the expansion of M to L', and M is a model of Δ, then M' will also be a model of Δ, now considered as a set of sentences in L' (not containing the predicate symbol G). On the other hand, any structure M' for L' that is a model of Δ will be an expansion of some structure M for L, where M is a model for Δ. Therefore, the set of expansions of models in L for Δ is the same as the set of models in L' for Δ. Using this fact, we get that (5.4) is equivalent to

(5.5) the assertion $WP(S,G(w)) \Rightarrow WP(S',G(w))$ holds in any model M' of Δ, M' a structure for L'.

This is finally the same as the fact that $WP(S,G(w)) \Rightarrow WP(S',G(w))$ is a logical consequence of Δ, i.e. (5.5) is equivalent to

$$\Delta \models WP(S,G(w)) \Rightarrow WP(S',G(w)).$$

This gives us the main theorem, on which proofs of refinement between descriptions will rest.

THEOREM 5.4. *Let* S *and* S' *be legal descriptions from* V *to* W, *where* V *and* W *are finite nonempty sets of variables. Let* L' *be an expansion of* L *that we get by adding a new* k*-place predicate symbol* G *to the nonlogical symbols of* L, *where* k *is the cardinality of* W. *Let* w *be a list of variables such that* $\tilde{w} = W$. *Then for any set* Δ *of sentences of* L, *we have that*

$$\Delta \models S \leq S' \quad \text{iff} \quad \Delta \models WP(S,G(w)) \Rightarrow WP(S',G(w)).$$

COROLLARY 5.5. (Proof rule for refinement). *Let* S *and* S', V *and* W, G *and* w *be as in Theorem* 5.4. *Then for any countable set* Δ *of sentences of* L.

$$\Delta \models S \leq S' \quad \text{iff} \quad \Delta \vdash WP(S,G(w)) \Rightarrow WP(S',G(w)).$$

PROOF. By Theorem 5.4 and the completeness of $L_{\omega_1\omega}$ (Lemma 2.2). □

We sat that $S \leq S'$ is *provable from* Δ, denoted $\Delta \vdash S \leq S'$, if we from Δ can prove $WP(S,G(w)) \Rightarrow WP(S',G(w))$, where G and w are as in Theorem 5.4. Corollary 5.5 then says that $\Delta \models S \leq S'$ iff $\Delta \vdash S \leq S'$.

COROLLARY 5.6. (Proof rule for equivalence). *Let* S *and* S', V *and* W, G *and* w *be as in Theorem* 5.4. *Then for any countable set* Δ *of sentences of* L,

$$\Delta \models S \approx S' \quad \text{iff} \quad \Delta \vdash WP(S,G(w)) \Leftrightarrow WP(S',G(w)).$$

Theorem 5.4 together with its corollaries provides us with a technique for proving refinement between descriptions. This technique is complete, i.e. if $S \leq S'$ is a semantic consequence of the countable set of sentences Δ, then there is a proof of $S \leq S'$ from Δ. (The completeness is of a rather weak kind, however, as the proofs that exist may be infinitely long.) The proof technique is also *sound*, i.e. if we succeed in proving $S \leq S'$ from Δ, then

$S \leq S'$ will indeed be a semantic consequence of Δ.

Another consequence of Theorem 4.3 and 5.2 is the following.

THEOREM 5.7. *Let* S *and* S' *be legal descriptions from* V *to* W, V *and* W *finite nonempty sets of variables. Let* M *be a structure for* L, *and let* Q *be any formula of* L, $var(Q) \subseteq W$. *If* $S \leq_M S'$, *then*

$$WP(S,Q) \Rightarrow WP(S',Q)$$

holds in M.

PROOF. Let $M = \langle D,I \rangle$. Assume that $S \leq_M S'$. By Theorem 4.3 we have that

$$wp(int_M(S,V),q) \Rightarrow wp(int_M(S',V),q) \quad \text{for any } q \in E_D(W).$$

Because $int_M(Q,W) \in E_D(W)$, we therefore get that

$$wp(int_M(S,V),int_M(Q,W)) \Rightarrow wp(int_M(S',V),int_M(Q,W)),$$

and using Theorem 5.2, we thus have that

$$WP(S,Q) \Rightarrow WP(S',Q)$$

holds in M. □

COROLLARY 5.8. *Let* S *and* S', V *and* W *and* Q *be as in Theorem* 5.7, *and let* Δ *be a set of sentences of* L. *If* $\Delta \models S \leq S'$, *then*

$$\Delta \models WP(S,Q) \Rightarrow WP(S',Q).$$

PROOF. Directly by Theorem 5.7. □

COROLLARY 5.9. *Let* S *and* S', V *and* W *and* Q *be as in Theorem* 5.7, *and let* Δ *be a countable set of sentences of* L. *If* $\Delta \vdash S \leq S'$, *then*

$$\Delta \vdash WP(S,Q) \Rightarrow WP(S',Q).$$

PROOF. Follows from Corollary 5.8, by the completeness of $L_{\omega_1\omega}$ and Corollary 5.5. □

Finally, we prove a sinple induction rule for iteration, which will be very useful later on.

LEMMA 5.10. *Let* Δ *be a countable set of sentences of* L. *Let* V *be a finite nonempty set of variables. Let* S *and* S' *be legal descriptions from* V *to* V, *and let* B *be a formula of* L, $\mathrm{var}(B) \subseteq V$. *Then the following holds:*

(i) *If* $\Delta \vdash (B*S)^n \leq S'$ *for* $n < \omega$, *then* $\Delta \vdash (B*S) \leq S'$.

(ii) $\Delta \vdash (B*S)^n \leq (B*S)$, *for any* $n < \omega$.

PROOF.

(i) Assume that

$$\Delta \vdash (B*S)^n \leq S' \quad \text{for any } n < \omega.$$

Let L' be an expansion of L with a new predicate symbol G with k places, where k is the number of variables in V, and let v be a list of distinct variables, $\tilde{v} = V$. The assumption then implies that

$$\Delta \vdash \mathrm{WP}((B*S)^n, G(v)) \Rightarrow \mathrm{WP}(S', G(v)), \quad \text{for } n < \omega.$$

Using the inference rule for infinite disjunction, Lemma 2.4, this gives us that

$$\Delta \vdash \bigvee_{n<\omega} \mathrm{WP}((B*S)^n, G(v)) \Rightarrow \mathrm{WP}(S', G(v)),$$

i.e.

$$\Delta \vdash \mathrm{WP}(B*S, G(v)) \Rightarrow \mathrm{WP}(S', G(v)),$$

by the definition of WP, thus giving

$$\Delta \vdash (B*S) \leq S',$$

as required.

(ii) Let L', G and v be as above. We have by the axiom for infinite disjunction, Lemma 2.5, that

$$\Delta \vdash \mathrm{WP}((B*S)^n, G(v)) \Rightarrow \bigvee_{i<\omega} \mathrm{WP}((B*S)^i, G(v)), \quad \text{for any } n < \omega.$$

Thus we have that

$$\Delta \vdash WP((B*S)^n, G(v)) \Rightarrow WP(B*S, G(v)), \quad \text{for any } n < \omega,$$

giving the required result

$$\Delta \vdash (B*S)^n \leq (B*S), \quad \text{for any } n < \omega. \quad \square$$

5.3. BASIC PROPERTIES OF WEAKEST PRECONDITIONS

DIJKSTRA [15] gives five basic properties of weakest preconditions for his guarded commands. If we let $S: V \to W$ be a legal description, and let w be a list of distinct variables, $\tilde{w} = W$, then the corresponding properties for descriptions are $(var(Q), var(Q') \subseteq W)$:

(1) $WP(S,false) \Leftrightarrow false$

(2) $\forall w(Q \Rightarrow Q') \Rightarrow (WP(S,Q) \Rightarrow WP(S,Q'))$

(3) $WP(S,Q \wedge Q') \Leftrightarrow WP(S,Q) \wedge WP(S,Q')$

(4) $WP(S,Q) \vee WP(S,Q') \Rightarrow WP(S,Q \vee Q')$ and

(5) If $Q_i \Rightarrow Q_{i+1}$ for $i = 0,1,\ldots$, where $Q_0, Q_1, \ldots$ are formulas of L, $var(Q_i) \subseteq W$ for $i < \omega$, then

$$WP(S, \bigvee_{i<\omega} Q_i) \Rightarrow \bigvee_{i<\omega} WP(S,Q_i).$$

The first four properties will hold for any description S in a given structure M, while the fifth property (*continuity*) in general only holds for descriptions of bounded nondeterminacy. A sufficient condition guaranteeing that the nondeterminacy of a description is bounded is that each basic description occurring in the description is *finite*. The basic description $x/y.Q$ is *finite in the structure* M if $M \models finite(x,Q)$, where $finite(x,Q)$ is the formula

$$\bigvee_{n<\omega} \forall x_0 x_1 \ldots x_n [\bigwedge_{0\leq i\leq n} Q[x_i/x] \Rightarrow \bigvee_{0\leq i<j\leq n} x_i = x_j].$$

The basic description $x/y.Q$ is *finite in the set of sentences* Δ if $\Delta \models finite(x,Q)$. Thus property(5) will holds for a description S in Δ if each basic description occurring in S is finite in Δ.

We will need slightly more general versions of properties (2) - (4). First we need to make a preliminary definition (used for property (2)). Let S be a legal description from V to W. We say that the variable z is *constant in* S if z belongs to both V and W, and in addition:

(i) if S is $x/y.Q$, then z does not belong to $\tilde{x}$, and

(ii) if S is either $(S';S'')$, $(S' \vee S'')$, $(B \to S' | S'')$ or $(B * S')$, then z is constant in S' and S".

LEMMA 5.11. *Let* S: V → W *be a legal description. Let* Q_n *be formulas of* L, $var(Q_n) \subseteq W$, *for* $n < \omega$. Then

(i) $\forall x(Q_0 \Rightarrow Q_1) \Rightarrow (WP(S,Q_0) \Rightarrow WP(S,Q_1))$, *provided every variable in* $W - \tilde{x}$ *is constant in* S,

(ii) $WP(S,\bigwedge_{n<\alpha} Q_n) \iff \bigwedge_{n<\alpha} WP(S,Q_n)$, $\alpha < \omega_1$

(iii) $\bigvee_{n<\alpha} WP(S,Q_n) \Rightarrow WP(S,\bigvee_{n<\alpha} Q_n)$, $\alpha < \omega_1$

hold in any structure M *of* L.

PROOF. Properties (ii) and (iii) are obvious generalizations of (3) and (4) above and will not be proved here. To prove property (i), let $M = \langle D,I \rangle$, and let $f = int_M(S,V) \in F_D(V,W)$. It is straightforward to prove by induction on the structure of S, that if the variable z is constant in S, then the following holds: for any proper states $s \in V_D$ and $s' \in W_D$, if $s' \in int_M(S,V)(s)$, then $s(z) = s'(z)$.

Now choose a proper state $s \in V_D$ such that

(1) $val_M(\forall x(Q_0 \Rightarrow Q_1),s) = tt$ and

(2) $val_M(WP(S,Q_0),s) = tt$.

By Theorem 5.2, we get from (2) that

$$wp(f,int_M(Q_0,W))(s) = tt.$$

Thus for any $s' \in f(s)$, we have that $int_M(Q_0,W)(s') = tt$, i.e. $val_M(Q_0,s') = tt$. By assumption (1), $val_M(Q_0,s\langle d/x\rangle) = tt$ implies $val_M(Q_1,s\langle d/x\rangle) = tt$ for any list d of elements in D, $\ell(d) = \ell(x)$. Because $var(Q_0) \subseteq W$ and $s(z) = s'(z)$ for $z \in W - \tilde{x}$, we have that $val_M(Q_0,s') = val_M(Q_0,s\langle s'(x)/x\rangle) = tt$, giving $val_M(Q_1,s\langle s'(x)/x\rangle) = tt$, and thus that $val_M(Q_1,s') = tt$. From this we then conclude that $wp(f,int_M(Q_1,W))(s) = tt$, as s' was arbitrarily chosen, and using Theorem 5.2 again, we then have that $val_M(WP(S,Q_1),s) = tt$, which

proves this case. □

5.4. REPLACEMENTS IN DESCRIPTIONS

We will here show that the refinement relation has a replacement property needed for top-down development of programs. The property in question is that replacing a subdescription of a description with a refinement will result in a refinement of the description as a whole. Top-down program development will be further discussed in the next chapter.

First let S_1 and S_1' be legal descriptions from V_1 to V_2, and let S_2 and S_2' be legal descriptions from V_2 to V_3, where V_1, V_2 and V_3 are finite nonempty sets of variables. Let Δ be countable, and assume that

(5.6) $\quad \Delta \vdash S_1 \leq S_1'$

and

(5.7) $\quad \Delta \vdash S_2 \leq S_2'$.

Let G be a new predicate letter of k places, and let L' be the expansion of L that we get by adding G to the nonlogical symbols of L. The number of variables in V_3 is assumed to be k. Let v be a list of distinct variables, $\tilde{v} = V_3$. From (5.7) we get that

$$\Delta \vdash WP(S_2,G(v)) \Rightarrow WP(S_2',G(v)).$$

Using the inference rule GN in $L_{\omega_1\omega}$ (Section 2.3), we then get that

$$\Delta \vdash \forall v'(WP(S_2,G(v)) \Rightarrow WP(S_2',G(v))),$$

where v' is a list of distinct variables, $\tilde{v} = V_2$. By the Lemma 5.1, v' contains each variable free in the formula quantified. We may therefore use Lemma 5.11(i), which gives us

$$\Delta \vdash WP(S_1,WP(S_2,G(v))) \Rightarrow WP(S_1,WP(S_2',G(v))).$$

On the other hand, using Corollary 5.9, noting that Δ is also a set of sentences in L', and the assumption (5.6), we get

$$\Delta \vdash WP(S_1,WP(S_2',G(v))) \Rightarrow WP(S_1',WP(S_2',G(v))).$$

Combining these last two results, we have

$$\Delta \vdash WP(S_1,WP(S_2,G(v))) \Rightarrow WP(S_1',WP(S_2',G(v))),$$

i.e.,

$$\Delta \vdash (S_1;S_2) \leq (S_1';S_2'),$$

which is the result we sought.

In a similar way we prove that

$$\Delta \vdash S_1 \leq S_1' \quad \text{and} \quad \Delta \vdash S_2 \leq S_2'$$

implies

$$\Delta \vdash (S_1 \vee S_2) \leq (S_1' \vee S_2')$$

and

$$\Delta \vdash (B \rightarrow S_1 | S_2) \leq (B \rightarrow S_1' | S_2').$$

The analogous result for iteration is derived as follows. Let V be a finite nonempty set of variables, and let S and S' be legal descriptions from V to V. Let B be a formula of L, var(B) $\subseteq$ V. Assume that

$$\Delta \vdash S \leq S'.$$

We first show that

(5.8) $$\Delta \vdash (B*S)^n \leq (B*S')^n$$

holds for any $n < \omega$. For $n = 0$ the situation is clear, as both descriptions are identical in this case (= abort). Assume that (5.8) holds for n, $n < \omega$. By the previous result, we will then have that

$$\Delta \vdash S;(B*S)^n \leq S';(B*S')^n,$$

using the assumption and the induction hypothesis. This then gives

$$\Delta \vdash (B \to S;(B*S)^n \mid \text{skip}) \leq (B \to S';(B*S')^n \mid \text{skip}),$$

i.e. we get that

$$\Delta \vdash (B*S)^{n+1} \leq (B*S')^{n+1}$$

holds. This shows that (5.8) holds for every $n < \omega$.

We now apply Lemma 5.10(ii) to get

$$\Delta \vdash (B*S')^n \leq (B*S') \quad \text{for any } n < \omega.$$

Combining this with (5.8), and using the fact that refinement is transitive, we get

$$\Delta \vdash (B*S)^n \leq (B*S'), \quad \text{for any } n < \omega.$$

We can now use Lemma 5.10(i) to get from this that

$$\Delta \vdash (B*S) \leq (B*S'),$$

which is the required result.

We summarize these results in the following theorem.

THEOREM 5.12. (Replacement). *Let* S: V → W *be a legal description, containing the subdescription* T: V' → W'. *Let* T': V' → W' *be a legal description, and let* S': V → W *be the description that results from* S, *when* T *in* S *is replaced with* T'. *For any countable set* Δ *of sentences, we then have that*

$$\Delta \vdash T \leq T'$$

implies

$$\Delta \vdash S \leq S'.$$

PROOF. The result follows by induction on the structure of S, using the results proved above. □

CHAPTER 6

STEPWISE REFINEMENT USING DESCRIPTIONS

In this chapter we want to show how to use descriptions in program development by stepwise refinement. We start by giving an example of the informal use of the technique in Section 6.1. This example is taken from DIJKSTRA [15], with some small changes.

In Section 6.2 we then outline the way in which the informal technique of stepwise refinement can be turned into a formal one, based on the use of descriptions. Having a formal development of a program makes it possible to use the proof rule for refinement to establish the correctness of the refinement steps. This in turn will give us a formal proof of the correctness of the final program. In this section we will show how to achieve top-down development and operational and representational abstraction and how to justify the use of program transformations when developing a program using descriptions.

In Section 6.3 we will introduce restricted forms of descriptions, *program descriptions and abstractions*, which are better suited for program development. We will compute the weakest preconditions for these using the rules for computing weakest preconditions for descriptions. Programs will be special kinds of program descriptions (essentially the guarded commands of DIJKSTRA [15]).

6.1. AN EXAMPLE OF THE USE OF STEPWISE REFINEMENT

To make things more concrete, and to show the kinds of refinement steps possible, we will first give an example of program construction using stepwise refinement. The example is taken from DIJKSTRA [15], pp. 65-67. We follow Dijkstra's treatment quite closely, but will carry the refinement process one step further in order to include an important kind of refinement step not used by Dijkstra in this example. We will later use this example again to show how our formalism of stepwise refinement works in practice.

The problem considered by Dijkstra is the following: let X and Y be integers, $X > 1$ and $Y \geq 0$. We are to construct a program that will establish the condition

$$R: z = X^Y,$$

without using the exponentiation operation in our program. Here z is an integer variable.

The first refinement made by Dijkstra makes use of an "abstract" variable h. The condition

$$P: h \cdot z = X^Y \wedge h \geq 1$$

will be kept invariant in the loop of the following program:

S_1: h,z := X^Y,1; {P has been established}
do $h \neq 1 \rightarrow$ squeeze h under invariance of P od
{R has been established}.

Here h,z := X^Y,1 is a simultaneous assignment statement, i.e. h is assigned the value X^Y and z is assigned the value 1 simultaneously. The do $h \neq 1 \rightarrow$ ... od construction is a loop; the statement ... is repeated as long as the condition $h \neq 1$ is true. The statement "squeeze h under invariance of P" specifies what remains to be done; we have to give a piece of program meeting this specification, i.e. a program that will decrease the value of the variable h in such a way that condition P remains true.

We have to check that this solution is correct, i.e. that S_1 really does establish the condition R. If the loop terminates, then P must hold, and as the loop only can terminate when h = 1, this means that R must hold upon termination (because $P \wedge h=1 \Rightarrow R$). To show that the loop really does terminate, we note that $h \geq 1$ holds initially, and will also hold after each iteration of the loop. On the other hand, as each iteration will decrease the value of h, the situation h = 1 must sooner or later occur, terminating the loop.

In the next step, the exponentiation operation is removed. Dijkstra introduces two new variables x and y, which are used to represent the value of h by the condition

$h = x^y$.

Instead of manipulating the variable h directly, the program will manipulate the variables x and y that represent the value of h. Observing that when $h = x^y$ and $x > 1$, we have

$$h \neq 1 \quad \text{iff} \quad y \neq 0,$$

we get the next refinement:

```
S2: x,y,z := X,Y,1; {P has been established}
    do y ≠ 0 → y,z := y-1,z•x {P has not been destroyed} od
    {R has been established}.
```

Essential use has here been made of the fact that P always holds prior to the execution of the statement in the loop. Finally, Dijkstra observes that the statement

```
do 2|y → x,y := x•x,y/2 od
```

will not change the value h represented by the variables x and y, and may therefore be inserted before the statement y,z := y-1,z•x, without affecting the correctness of the program (2|y tests whether y is divisible by 2). This gives the refinement

```
S3: x,y,z := X,Y,1;
    do y ≠ 0 → do 2|y → x,y := x•x,y/2 od;
               y,z := y-1,z•x
    od,
```

yielding a considerable speed up of the program, as compared to S_2.

We will make an additional refinement of this, by noting that after each execution of the statement in the inner loop, the condition $y \neq 0$ must hold, if it was true on entry to the inner loop. Therefore the two nested loops may be fused into one, giving the last refinement

```
S4: x,y,z := X,Y,1;
    do y ≠ 0 → if 2|y → x,y := x•x,y/2
               □ ~2|y → y,z := y-1,z•x fi
    od.
```

Here if ... fi is a conditional statement, selecting to execute the statement for which the test is true. This last refinement is simpler in that it only contains a single loop, as compared to S_3, which contains two nested loops. It is, however, less efficient than S_3, because in some situations the test $y \neq 0$ is performed unnecessarily.

As can be seen from the example, stepwise refinement combines two different principles of program development: *top-down development* and *optimizing transformations*. Top-down development of programs proceeds by implementing specifications, i.e. giving algorithms that meet stated criteria. This is the case in the example for the first refinement S_1, which is required to satisfy the specification given, i.e. to establish the condition R. As another example, the statement "y,z := y-1,z•x" is required to satisfy the specification "squeeze h under invariance of P", given the representation of h by x and y, and the fact that P holds prior to this specification in S_1.

The refinement of S_1 to S_2 is an example of the use of *representational abstraction,* i.e. the data structure (the variable h) used in S_1 is an abstraction of the data structure (the variables x and y) used in S_2. The refinement of S_2 to S_3 exploits the fact that this representation of h by x and y is not unique. Finally the refinement of S_3 to S_4 can be seen as an application of a special program transformation rule (as noted above this is not strictly speaking an optimizing transformation).

The application of both top-down development and optimizing transformations makes stepwise refinement very flexible as a programming technique. The top-down approach allows a programmer to move from a higher to a lower level of abstraction in constructing the program, and to concentrate on only part of the program when making a refinement step. Optimizing transformations are again useful in removing inefficiencies introduced by the top-down approach when the interaction between different program parts could not be considered.

6.2. CORRECT REFINEMENTS USING DESCRIPTIONS

In this section we will discuss principles for developing programs in such a way that the correctness of the final program can be formally proved. We will try to stay as close as possible to the informal technique for program development shown in the preceding section, while still staying in the framework of refinement between descriptions developed in the preceding chapter.

6.2.1. Top-down development

The fact that the transitivity of refinement justifies a stepwise construction of the final program was already noted in the introduction. Thus, if we have the development sequence

$$S_0, S_1, \ldots, S_{n-1}, S_n$$

where S_0 is the initial specification and S_n is the final program, and if each refinement step in this sequence is correct, i.e. if

$$S_i \leq S_{i+1}$$

holds for $i = 0,1,\ldots,n-1$, then transitivity gives us that

$$S_0 \leq S_n,$$

i.e. S_n satisfies specification S_0.

Stepwise refinement is, however, more than this. It also makes use of the idea of top-down development, i.e. the idea that one can concentrate on a subcomponent of the program, refining this independently from the rest of the program and then finally replace the subcomponent with its refinement.

The fact that this is allowed with descriptions too is given by Theorem 5.12. Let S be a description with an occurrence of the subdescription T, i.e.

$$S = \ldots T \ldots$$

and assume that we have a refinement T' of T, i.e.

$$T \leq T'.$$

Let S' be the description S with T replaced by T', i.e.

$$S' = \ldots T' \ldots .$$

By Theorem 5.12, this means that

$$S \leq S',$$

i.e. the replacement of T with T' in S is a correct refinement step.

6.2.2. The assignment statement

The assignment statement is usually chosen as the basic construct in programming languages. Although the language of descriptions does not contain assignment statements, the effect of an assignment statement is easily achieved. Consider e.g. the assignment statement

$$x := x + y.$$

The same effect can be achieved with the description

$$z/\langle\rangle.(z = x+y);$$
$$x/z.(x = z),$$

where z is a new variable, not occurring in the context where the assignment statement is used. Multiple assignments can be handled in the same way. A partial assignment statement such as

$$x := x/y$$

would again be expressed by the description

$$z/\langle\rangle.(z = x/y \wedge y \neq 0);$$
$$x/z.(x = z).$$

This description will not terminate when y = 0 initially, i.e. we use non-termination as an indication of an error.

Note that it would not have been correct to express the first assignment statement as

$$x/<>.x = x + y,$$

because this would have the effect of setting x to some value satisfying the equation x = x+y. For $y \neq 0$ this equation has no solution x, while for y = 0 any value of x would do. This is thus an example of a description which is both partial and nondeterministic, and where the nondeterminism is in fact unbounded.

In the next section we will show that not only the assignment statement but also the if ... fi and the do ... od constructions are expressible using descriptions, i.e. the programs in the previous section can be expressed as descriptions.

6.2.3. Replacements in context

The top-down property of descriptions guarantees that certain kinds of replacements are always allowed. There are, however, replacements that lead to refinements of the original description, but which cannot be justified by the top-down property alone. Consider the following example. Let S be the description

$$S = (x \geq 0 \rightarrow x := |x| + 1 \mid x := x*x).$$

We want to replace the assignment statement 'x:=|x|+1' with the statement 'x:=x+1'. This replacement is obviously correct, because the first assignment statement will only be executed when $x \geq 0$, in which case the assignment statement 'x:=x+1' has the same effect. However,

$$x := |x| + 1 \leq x := x + 1$$

does not hold, because for $x < 0$ they give different results. What we have here is a replacement that is correct in the context that it occurs, but which is not generally correct, i.e. it is not correct in every context.

To handle this kind of replacement, we use a special class of descriptions called *assertions*. An assertion {R} denotes the description

$$<>/<>.R,$$

where R is some formula. It acts as a partial skip statement, i.e. if the initial state satisfies R, then the assertion has no effect, but if the initial state does not satisfy R, it acts as an abort statement, i.e. the statement will not terminate.

Returning to the example, what we can prove is that 'x:=x+1' is a refinement of 'x:=|x|+1' for initial states satisfying $x \geq 0$, i.e. we can prove that

$$\{x \geq 0\};\ x := |x| + 1 \leq x := x + 1$$

holds. Therefore we should first prove that

$$S \leq (x \geq 0 \rightarrow \{x \geq 0\};\ x := |x| + 1 \mid x := x*x)$$

holds, and then use the replacement Theorem 5.12 to get that

$$(x \geq 0 \rightarrow \{x \geq 0\};\ x:= |x| + 1 \mid x := x*x)$$

$$\leq (x \geq 0 \rightarrow x := x + 1 \mid x := x*x).$$

Transitivity then gives the required result, i.e.

$$S \leq (x \geq 0 \rightarrow x := x + 1 \mid x := x*x).$$

The general situation is as follows. We have a description S with an occurrence of the description T in it, i.e.

$$S = \ldots\ T\ \ldots\ .$$

We want to replace T with T'. If $T \leq T'$ holds, this can be done immediately by Theorem 5.12. Otherwise we try to find an assertion {R} such that

$$S \leq S',$$

where

$$S' = \ldots\{R\};\ T\ldots\ .$$

If we can prove that

$$\{R\};\ T \leq T',$$

we have by Theorem 5.12 that

$$S' \leq S'',$$

where

$$S'' = \ldots\ T'\ \ldots\ .$$

Transitivity then gives the desired result, i.e.

$$S \leq S''.$$

6.2.4. Program transformation rules

A program transformation rule will in general give for each description S of a certain form a transformed description $\tau(S)$. If certain assumptions about S are satisfied, then the transformation will be correct, i.e.

$$S \leq \tau(S)$$

will hold.

In the previous example, we could have used the program transformation rule

$$\{R\};(B \rightarrow S_1 \mid S_2) \leq (B \rightarrow \{R \wedge B\};S_1 \mid \{R \wedge \sim B\};S_2)$$

to justify the introduction of the assertion $\{x \geq 0\}$ into the program.

Another simple transformation is

$$\{R\};(B \rightarrow S_1 \mid S_2) \leq S_1,$$

which holds if $R \Rightarrow B$.

Program transformation rules correspond to derived rules of inference in the logic $L_{\omega_1\omega}$. They are of the general form

$$\frac{\Phi}{S \leq \tau(S)},$$

where Φ is the set of assumptions made. The soundness of such a rule can be shown by deriving $S \leq \tau(S)$ in $L_{\omega_1\omega}$ from the assumptions Φ. Program transformation rules of this kind will be treated extensively in Chapter 7, where their correctness will be shown in the manner suggested. These program transformations will be concerned with the introduction of assertions into descriptions (Section 7.3), changing control structures in a description (Section 7.4) and the use of representational abstraction (Section 7.5).

6.2.5. Operational abstraction

The way in which the assignment statement was expressed using a description can be generalized to a *nondeterministic assignment*. An example of a nondeterministic assignment is

$$x := x'.\ (|x^2 - x'| < e).$$

The intended effect of this is that the variable x is assigned some new value x' such that

$$|x^2 - x'| < e$$

will hold, without changing the values of the other variables. Thus the effect is roughly to perform the operation $x := x^2$ with precision e. The operation is both nondeterministic (any value x' in the range $x^2-e < x' < x^2+e$ will do) and partial (it is not defined for $e \leq 0$).

This nondeterministic assignment can be expressed by the description

$$z/<>.(|x^2-z| < e);$$
$$x/z.(x = z),$$

where z as before is a new variable, not used in the context where the nondeterministic assignment occurs.

A procedure is usually specified by giving its entry and exit conditions. Thus a procedure for squaring x with precision e would have the entry condition

$$e > 0,$$

and the exit condition

$$|x^2 - x'| < e,$$

with x' denoting the new value of x, while x itself stands for the initial value of x. In addition, we would like to state that only x may be changed by the procedure (thus e.g. forbidding the procedure to change e). The fact that the description S satisfies these entry and exit conditions can be expressed by

$$\{e > 0\};\ x := x'.(|x^2 - x'| < e) \leq S. \tag{6.1}$$

This states that S will compute the square of x with precision e for initial states in which e > 0 holds.

Operational abstraction is thus achieved by using the procedure specification

$$\{e > 0\};\ x := x'.(|x^2 - x'| < e)$$

as such in a certain stage of the program development. At a later stage an implementation S satisfying this specification, i.e. satisfying (6.1) above, can be given. Replacing the specification with S is then allowed by Theorem 5.12. This scheme allows us to use parameterless procedures in program development, without having to introduce names for these procedures. (Recursive procedures cannot, of course, be handled in this manner.)

In the next chapter (Section 7.2) we will give special proof rules for proving the correctness of procedure implementations, i.e. for proving refinements of the type in (6.1). We will there also show that these special proof rules are derivable from the general proof rule for refinement.

6.2.6. Representational abstraction

An example of representational abstraction was already given in the preceding section, in the transition from program S_1 to program S_2. Another example is the following.

Consider a program which uses a set V of variables. Let A be a variable of S which only takes small sets of integers as values (small means here that the sets have at most 100 elements). We want to represent the variable A by the new variables B and k, where B is to be an integer array with indices running from 1 to 100 and k an integer in the range from 0 to 100.

In order to specify the way in which the variables B and k are to represent the variable A, we first have to indicate those value combinations of B and k that are meaningful, i.e. which represent some small set of integers. This is done by giving a condition I that B and k must satisfy if they are to represent anything. In this case we give the condition

I(B,k): B is an integer array [1..100] and
k is an integer in range 0..100.

We also have to indicate what small set of integers B and k represent when they satisfy the condition I(B,k). This is done by giving a function t, which assigns to each value combination B and k the set of integers represented by B and k. In this case we give

$$t(B,k) = \{B[i] \mid 1 \leq i \leq k\} .$$

Here the function t is the *abstraction function* and the condition I the *concrete invariant,* introduced in HOARE [25] as an aid to proving the correctness of data representation. The example here is also taken from this reference, although Hoare uses a stronger concrete invariant then the one given here.

We now have two different data spaces, the "abstract" data space V in which the variable A occurs, and the "concrete" data space $W = (V - \{A\}) \cup \{B,k\}$, in which A is replaced by the variables B and k. The transition from the concrete data space to the abstract data space can be given by a description

$$\alpha = A/B,k.(A = t(B,k) \wedge I(B,k)).$$

This transition is defined when B and k satisfy the condition I, and it will assign to the variable A the value represented by the variables B and k. On the other hand, the transition from the abstract data space to the concrete data space can be given by the description $\beta : V \to W$, defined by

$$\beta = B,k/A.(A = t(B,k) \wedge I(B,k)).$$

This will assign to the variables B and k some values which represent the value of A. It will be defined if A has a representation using B and k, i.e. if the value of A is some small set of integers.

The descriptions α and β are each others inverses. Note that description α is deterministic while description β is not. This means that there is more than one way to represent a given small set using B and k, but that each B and k satisfying the condition I will represent a unique small set (in fact, there are infinitely many different ways of representing a small set with less than 100 elements, because the choice of B[i] for i > k does not matter).

Consider now the problem of finding a refinement of S where the variable A is represented by the variable B and k. This can be expressed as follows: find a description $S' : W \to W$ such that

$$(6.2) \qquad \{R\}; S \leq \beta; S'; \alpha$$

holds. Here R is a condition that guarantees that A has a value that can be represented by B and k. In this case we would have

R(A): A is a small set of integers.

The assertion {R} is necessary to restrict the refinement to those initial states for which β is defined. It is possible that S could also be defined for initial states that do not satisfy R (e.g. S could be defined for any sets of integers, and not only for small sets).

The refinement (6.2) can be operationally interpreted as follows: for initial states satisfying R, the effect of S can be achieved by first finding some representation of A using B and k, then using S' to get a final state by manipulating the variables B and k, and then setting A to the value represented by the final values of B and k.

An S' satisfying (6.2) can now be constructed in the following way. We may either simply invent an S' satisfying (6.2), and then the problem is solved. Or if S is of the form $(S_1;S_2)$, $(S_1 \vee S_2)$, $(B \to S_1 | S_2)$ or $(B*S_1)$, where S_1, S_2: $V \to V$, we can reduce the problem to the corresponding subproblems for S_1 and S_2. Consider as an example the case

$$S = S_1;S_2.$$

As a first step we prove that

$$\{R\};S \leq \{R\};S_1;\{R\};S_2$$

using some transformation rules for introducing the assertions. Then we solve the subproblems of finding S_1' and S_2' that satisfy

$$\{R\};S_1 \leq \beta;S_1';\alpha$$

and

$$\{R\};S_2 \leq \beta;S_2';\alpha.$$

Using the replacement property (Theorem 5.12), we then have that

$$\{R\};S_1;\{R\};S_2 \leq (\beta;S_1';\alpha);(\beta;S_2';\alpha).$$

Finally, it can be shown that the transformation rule

$$(6.3) \qquad (\beta;S_1';\alpha);(\beta;S_2';\alpha) \leq \beta;(S_1';S_2');\alpha$$

is always correct, provided α and β satisfy certain properties (to be given later, in Section 7.5). Transitivity of refinement then gives us the desired result, i.e.

$$\{R\};S \leq \beta;S';\alpha,$$

where

$$S' = S_1';S_2' .$$

The other cases can be treated in a similar way. Transformation rules of the form (6.3) will be the subject of Section 7.5 in the next chapter.

An important special case occurs when the program S uses the variable A as a "temporary" variable, i.e. S will initialize the variable A to some value, and it does not depend on the initial value of A. In this case we introduce the description β_0: V → W, defined by

$$\beta_0 = \text{B,k/A. true.}$$

This description will assign arbitrary values to B and k. The requirement to be put on S': W → W is now that

$$S \leq \beta_0;S';\alpha$$

holds. The restriction R can be dropped here because β_0 is always defined.

An S' can be found by the same technique as above. If we assume that $S = S_1;S_2$, we first prove that

$$S \leq S_1;\{R\};S_2.$$

Then we solve the problems of finding S_1' and S_2' satisfying

$$S_1 \leq \beta_0;S_1';\alpha$$

and

$$\{R\};S_2 \leq \beta;S_2';\alpha\ .$$

By replacement we again get that

$$S_1;\{R\};S_2 \leq (\beta_0;S_1';\alpha);(\beta;S_2';\alpha).$$

Finally we use a program transformation rule that gives

$$(\beta_0;S_1';\alpha);(\beta;S_2';\alpha) \leq \beta_0;(S_1';S_2');\alpha\ .$$

By transitivity, we then have the desired result, i.e.

$$S \leq \beta_0;S';\alpha,$$

where $S' = S'_1;S'_2$.

6.2.7. Safe program development

The refinement relation is based on strong termination rather than on the natural notion of termination, which includes both strong and weak termination. We would like to restrict the refinement relation to cases where weak termination cannot occur. In such cases the natural notion of termination coincides with strong termination, and one does not have to bother about the distinction between these two. We refer to program development where the problems of weak termination can be ignored as *safe program development*.

The obvious way to exclude weak termination would be to restrict all basic descriptions to be finite, thus guaranteeing that all descriptions are of bounded nondeterminacy. This would, however, restrict the technique for changing data representation to only allow finite transitions between abstract and concrete data spaces. More precisely, an abstract value could only be represented by a finite number of different concrete values. This seems to be unnecessarily restrictive (the abstraction function t(B,k) above does not e.g. satisfy this restriction). We will therefore choose a slightly less restrictive way of guaranteeing safe program development.

We will in the next section define a subset of descriptions which are guaranteed to be of bounded nondeterminacy. A special notation will be introduced for descriptions in this subset, making them look very much like ordinary program statements. We will call these descriptions *program descriptions*. They will essentially be the guarded commands of Dijkstra, extended with operational abstraction, assertions and a simple block structure, allowing the introduction of local variables. Weak termination will be excluded by requiring the nondeterministic assignment statements to make their choices from a finite set of possibilities only, and by requiring that local variables in blocks are properly initialized.

Besides program descriptions, we also will have *abstractions* of program descriptions. These are descriptions of the form $\beta;S;\alpha$ where β and α are transitions between abstract and concrete state spaces of the kind described above, and S ia a program description. Weak termination cannot occur for descriptions of this form either, although the descriptions can be of unbounded nondeterminacy (the transitions α and β are not required to be finite).

We will further restrict ourselves to only consider refinements of the form $S \leq S'$, where S is a program description and S' is either a program description or an abstraction of a program description. In the next chapter we will give a number of inference rules by which such refinements can be derived. The correctness of these inference rules will be shown using the general proof rule for refinement. We will also show how the example program of Section 6.1 can be derived with the aid of these rules of inference.

6.2.8. Related work

The approach to stepwise refinement presented above is new, as far as we know. Related ideas have, however, been presented before. Thus KATZ and MANNA [28] contains a similar technique of using assertions to collect information about the context of a program part. The nondeterministic assignment has been used previously in HAREL et al. [22] in the extension given for Hoare's axiomatic system, and more extensively in BAUER [6]. The formalism of representational abstraction given here is clearly inspired by the *abstract data type* facility first discussed in HOARE [25], and provided in a number of new programming languages (see e.g. WULF et al. [46], WIRTH [45], LAMPSON et al. [31] and LISKOV et al. [32]). Representational abstraction is, however, a more general (and less structured) concept than the abstract data types, permitting e.g. two or more abstract variables to share the same concrete variables for representation. The way in which representational abstraction is handled here is somewhat similar to the handling of abstraction in BURSTALL and DARLINGTON [10] or the concept of simulation between programs defined in MILNER [36].

6.3. PROGRAM DESCRIPTIONS

We define the set Vr of *program variables* by

$$Vr = \{v_n \mid n = 2k \text{ for some } k < \omega\}.$$

The set of *marked variables* Vr' is defined by

$$Vr' = \{v_n \mid n = 2k+1 \text{ for some } k < \omega\}.$$

For each variable v_n in Vr, v_n' denotes the *corresponding* marked variable v_{n+1} in Vr'. For any set U (list x) of program variables, U' (x') is the set (list) of corresponding marked variables.

Let V be a finite nonempty set of program variables. The *program descriptions in* V form a subset of the legal descriptions from V to V. We define them below, at the same time giving a notation for them. We assume throughout that a language L and a set of axioms Δ has been fixed.

6.3.1. Assertions

Let Q be a formula of L, $var(Q) \subseteq V$. Then the *assertion*

$$\{Q\} =_{df} {<>}/{<>}.Q$$

is a program description in V. As special cases of assertions we have the *skip statement*

$$skip =_{df} \{true\},$$

and *abort statement*

$$abort =_{df} \{false\}.$$

The skip and abort statement have the same meaning here as they have in DIJKSTRA [15] and in Section 5.1 above.

The weakest preconditions for these constructs are as follows:

$$WP(\{Q\},R) \Longleftrightarrow Q \wedge R,$$

$$WP(skip,R) \Longleftrightarrow R,$$

$$WP(abort,R) \Longleftrightarrow false.$$

This follows directly by computation. We have

$$WP(\{Q\},R) = WP({<>}/{<>}.Q,R)$$

$$\Leftrightarrow Q \wedge (Q \Rightarrow R)$$

$$\Leftrightarrow Q \wedge R.$$

We then have that

$$WP(skip,R) \iff true \wedge R \iff R$$

and

$$WP(abort,R) \iff false \wedge R \iff false.$$

The effect of the assertion was already explained in the previous section.

6.3.2. Assignment

Let Q be a formula of L and x a list of distinct variables in V, where $var(Q) \subseteq V \cup \tilde{x}'$. Assume that $\Delta \vdash finite(x',Q)$. Then the *(finite nondeterministic) assignment*

$$x := x'.Q \underset{df}{=} x'/<>.Q;\ x/x'.(x=x')$$

is a program description in V. The effect of the assignment statement is to assign new values to the variables in the list x, so that condition Q becomes true. The marked variables x' in Q stand for the new values assigned to x, while x itself stand for the old values. No other variables are affected by this statement.

A special case of the assignment is the *assignment statement*

$$x := t \underset{df}{=} x := x'.(x'=t).$$

where x is a list of variables of V and t is a list of terms of L, $\ell(x) = \ell(t)$ and $var(t_i) \subseteq V$ for $i = 1,\ldots,\ell(t)$.

The weakest precondition for the assignment and the assignment statement will be

$$WP(x:=x'.Q,R) \iff \exists x'Q \wedge \forall x'(Q \Rightarrow R[x'/x])$$

and

$$WP(x:=t,R) \iff R[t/x].$$

For the assignment, the weakest precondition is computed as follows:

$$WP(x:=x'.Q,R) = WP(x'/<>.Q,\ WP(x/x'.x=x',R)).$$

We have

$$WP(x/x'.x=x',R) = \exists x(x=x') \wedge \forall x(x=x' \Rightarrow R)$$

$$\Leftrightarrow \text{true} \wedge R[x'/x] \text{ (by Lemma 2.6)}$$

$$\Leftrightarrow R[x'/x].$$

Thus

$$WP(x:=x'.Q,R) \Longleftrightarrow WP(x'/x.Q,R[x'/x])$$

$$\Longleftrightarrow \exists x'Q \wedge \forall x'(Q \Rightarrow R[x'/x]).$$

For the assignment statement we have

$$WP(x:=t,R) = WP(x:=x'.x'=t,R)$$

$$\Leftrightarrow \exists x'(x'=t) \wedge \forall x'(x'=t \Rightarrow R[x'/x])$$

$$\Leftrightarrow \text{true} \wedge R[t/x]$$

$$\Leftrightarrow R[t/x].$$

The angular brackets for lists will usually be dropped in assignments and assignment statements in examples. However, we will still write <> for the empty list of variables.

6.3.3. Composition

We have composition for program descriptions in the same way as for descriptions. Parenthesis may be dropped, by agreeing that $S_1;S_2;\ldots;S_{n-1};S_n$ stands for $(S_1;(S_2;(\ldots;(S_{n-1};S_n)\ldots)))$.

6.3.4. Nondeterministic selection

Let $S_1,\ldots,S_n$ be program descriptions in V, and let $B_1,\ldots,B_n$ be formules of L, such that $var(B_i) \subseteq V$ for $i = 1,\ldots,n$, $n \geq 1$. The *nondeterministic selection*

$$\underline{\text{if}}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{fi}}$$

is then a program description in V. It is defined as follows:

$$\underline{\text{if}}\ B_1 \to S_1\ \underline{\text{fi}} = (B_1 \to S_1 \mid \text{abort}),$$

$$\underline{\text{if}}\ B_1 \to S_1 \,\square B_2 \to S_2\ \underline{\text{fi}}$$

$$= (B_1 \wedge \sim B_2 \to S_1 \mid$$

$$(B_2 \wedge \sim B_1 \to S_2 \mid \underline{\text{if}}\ B_1 \wedge B_2 \to S_1 \vee S_2\ \underline{\text{fi}})).$$

$$\underline{\text{if}}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{fi}}$$

$$= \underline{\text{if}}\ B_1 \to S_1$$

$$\square\ B_2 \vee \ldots \vee B_n \to \underline{\text{if}}\ B_2 \to S_2 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{fi}}$$

$$\underline{\text{fi}},$$

$$\text{for } n > 2.$$

A reasonable amount of computation will show that the weakest precondition for the nondeterministic selection is

$$WP(\underline{\text{if}}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{fi}},\ R)$$

$$= \bigvee_{1\leq i\leq n} B_i \wedge \bigwedge_{1\leq i\leq n} (B_i \Rightarrow WP(S_i,R)).$$

6.3.5. Nondeterministic iteration

Let $S_1,\ldots,S_n$ be program descriptions in V, and let $B_1,\ldots,B_n$ be formulas of L, such that $var(B_i) \subseteq V$ for $i = 1,\ldots,n$, $n \geq 1$. Then the *nondeterministic iteration*

$$\underline{do}\ B_1 \to S_1 \,\square \ldots \square B_n \to S_n\ \underline{od}$$

$$=_{df} ((B_1 \vee \ldots \vee B_n)\ *\ \underline{if}\ B_1 \to S_1 \,\square \ldots \square B_n \to S_n\ \underline{fi})$$

is a program description in V.

The weakest precondition for nondeterministic iteration is

$$WP(\underline{do}\ B_1 \to S_1 \,\square \ldots \square B_n \to S_n\ \underline{od},\ R)$$

$$= \bigvee_{n<\omega} WP(\underline{do}\ B_1 \to S_1 \,\square \ldots \square B_n \to S_n\ \underline{od}^n,\ R)$$

where

$$\underline{do}\ B_1 \to S_1 \,\square \ldots \square B_n \to S_n\ \underline{od}^n$$

$$=_{df} ((B_1 \vee \ldots \vee B_n)\ *\ \underline{if}\ B_1 \to S_1 \,\square \ldots \square B_n \to S_n\ \underline{fi})^n,$$

for $n \geq 0$.

Noting that

$$(B \to S_1 \,|\, S_2) \approx \underline{if}\ B \to S_1 \,\square \sim B \to S_2\ \underline{fi},$$

we find that

$$\underline{do}\ B_1 \to S_1 \,\square \ldots \square B_n \to S_n\ \underline{od}^0 = \text{abort}$$

and

$$\underline{\text{do}}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{od}}^n$$

$$\approx \underline{\text{if}}\ BB \to \underline{\text{if}}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{fi}};$$

$$\underline{\text{do}}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{od}}^{n-1}$$

$$\square\ \sim BB \to \text{skip}$$

$$\underline{\text{fi}},$$

for $n > 0$.

where BB denotes the condition $B_1 \vee \ldots \vee B_n$.

6.3.6. Blocks

Let S be a program description in $V \cup \tilde{x}$, $\tilde{x} \cap V = \emptyset$. Then the *block*

$$\underline{\text{beg}}\ x\text{: } S\ \underline{\text{end}} =_{df} x/<>.\text{true};S;<>/x.\text{true}$$

is a program description in V. The weakest precondition for this is easily computed to be

$$WP(\underline{\text{beg}}\ x\text{: } S\ \underline{\text{end}}, R) = \forall x WP(S,R).$$

The purpose of the block construct is to allow the introduction of local variables. It is more restrictive than the usual block construct, in that redeclaration of local variables is not allowed.

We will require that all local variables of a block are properly initialized before their values are referred to. We will not, however, give any specific schemes according to which this condition is to be guaranteed. The simplest way would of course be to require that each block begins with an explicit assignment of values to the local variables, but this could be too restrictive. Other possibilities have been discussed by DIJKSTRA [15] and by DE BAKKER [12].

If the use of uninitialized variables was allowed in blocks, one could simulate nondeterministic assignments which are not finite. For instance, the following block would have the effect of setting x to any value:

beg y: x := y end

Thus proper initialization of local variables is required to prevent weak termination from occurring.

The program descriptions are now the descriptions generated by the constructs above. The *programs* are generated by these same constructs, when restricted so that we only allow the skip and abort statement, the assignment statement, composition, selection and iteration with boolean expressions as guards and blocks. Programs are thus the *guarded commands* of DIJKSTRA [15], plus the block construction. The weakest preconditions for programs are also the same as those given by Dijkstra, except for the weakest precondition for the nondeterministic iteration, which, however, is equivalent to the weakest precondition given by Dijkstra.

Program specifications are special kinds of program descriptions. A program specification, giving the entry condition P and the exit condition Q and allowing only the variables in x to be changed, is expressed as the program description

{P}; x := x'.Q.

or, equivalently,

if P → x:=x'.Q fi.

No special notation will be introduced for program specifications.

NOTE: The burden of checking condition $\Delta \vdash \text{finite}(x',Q)$ for assignments x := x'.Q in program descriptions can be greatly reduced, if we assume that we have available a set of standard formulas which are finite in Δ, i.e. formulas B which satisfy condition $\Delta \vdash \text{finite}(x',B)$. With these we could restrict ourselves to assignments of the form $x := x'.(B \wedge Q)$, where B is some standard finite formula. The finiteness of this assignment does not have to be separately verified, as it is a consequence of the finiteness of B. We could even introduce a special notation for these, e.g.

$$x := x'(B).Q =_{df} x := x'.(B \wedge Q),$$

In the domain of integers, the finite subranges would be typical examples of standard finite formulas, E.g. the assignment

$$u := u'(1 \leq u' \leq v).Q(u,u',v)$$

will be finite, no matter what Q is.

Finally we will define abstractions of program descriptions, to be used when changing the data representation of a program description.

6.3.7. Abstraction

Let x/y.Q be an atomic description from V to W, where $\tilde{x} \cap V = \emptyset$. Let Q be the formula $y = t \wedge I$ where t is a list of terms in L and I is a formula of L, $var(t_i) \subseteq W$ for $i = 1,\ldots,\ell(t)$, $var(I) \subseteq W$ and $\ell(t) = \ell(y)$. Let S be a program description in W. Then

$$\underline{rep}\ x/y.Q\colon S\ \underline{per} =_{df} x/y.Q;\ S;\ y/x.Q$$

and

$$\underline{beg}\ x/y.Q\colon S\ \underline{per} =_{df} x/y.true;\ S;\ y/x.Q$$

are *abstractions* in V of the program description S in W.

The weakest preconditions for these are:

$$WP(\underline{rep}\ x/y.Q\colon S\ \underline{per},\ R)$$

$$\Leftrightarrow \exists x(y=t\wedge I) \wedge \forall x(y=t\wedge I \Rightarrow WP(S, I\wedge R[t/y]))$$

and

$$WP(\underline{beg}\ x/y.Q\colon S\ \underline{per},\ R) \Longleftrightarrow \forall x WP(S, I \wedge R[t/y]).$$

The computation of these goes as follows. We compute first

$$WP(y/x.y=t \wedge I, R) = \exists y(y=t \wedge I) \wedge \forall y(y=t \wedge I \Rightarrow R).$$

We have by Lemma 2.6 that

$$\exists y(y=t \wedge I) \Longleftrightarrow I[t/y] \Longleftrightarrow I,$$

because y is not free in I. On the other hand, by axiom Q1 and Lemma 2.6,

$$\forall y(y=t \wedge I \Rightarrow R) \Longleftrightarrow \forall y(I \Rightarrow (y=t \Rightarrow R)) \Longleftrightarrow I \Rightarrow \forall y(y=t \Rightarrow R)$$
$$\Longleftrightarrow I \Rightarrow R[t/y],$$

for the same reason. Thus we get that

$$WP(y/x.y=t \wedge I,R) \iff I \wedge (I \Rightarrow R[t/y]) \iff I \wedge R[t/y].$$

Thus the result will follow by computing

$$WP(\underline{rep}\ x/y.y=t \wedge I:\ S\ \underline{per},\ R)$$

$$\iff WP(x/y.y=t \wedge I,\ WP(S,I \wedge R[t/y]))$$

and

$$WP(\underline{beg}\ x/y.y=t \wedge I:\ S\ \underline{per},\ R)$$

$$\iff WP(x/y.true,\ WP(S,I \wedge R[t/y])).$$

The purpose of an abstraction is to allow the state space to be changed. The abstraction T of S in W, T an abstraction in V,

$$T = \underline{rep}\ x/y.y=t \wedge I:\ S\ \underline{per}$$

will change the state space by replacing the variables y in V with new variables x that represent y by the equation

$$y_i = t_i, \quad \text{for } i = 1,\ldots,\ell(y).$$

Here t_i are terms whose values depend on the variables in x and possibly on some other variables in W. There may be more than one choice of values for the variables in x that will represent y. The values chosen for x must, however, satisfy the condition I. After this the description S is executed. Finally, the variables in y are assigned the values represented by the new values of x (and possibly by some other variables in W). All in all, the effect of the abstraction T is to manipulate the variables in y by manipulating a representation x of these variables.

The abstraction

$$\underline{beg}\ x/y.y=t \wedge I:\ S\ \underline{per}$$

is used to initialize the variables y by initializing the variables x which represent the variables y.

CHAPTER 7

FORMAL DEVELOPMENT OF PROGRAMS

In this final chapter we show how programs can be formally derived using the machinery developed above. The use of program descriptions makes formal proofs of the correctness of derivations possible. The general proof rule for refinement can in principle be used for establishing the correctness of the individual refinement steps in the derivation. In practice, however, this is not very convenient and we need stronger proof rules for handling the different kinds of refinement steps commonly occurring in program development.

In Section 7.1 we show how to derive the example program of Section 6.1 in a formal way using program descriptions. This derivation makes use of a number of stronger proof rules by which the correctness of the refinement steps can be proved. These proof rules will be formulated in the following sections. Thus Section 7.2 gives proof rules for proving the correctness of precedure implementations. Section 7.3 will give examples of transformation rules by which assertions can be introduced into descriptions. Section 7.4 gives an example of a transformation rule by which the control structure of a program description can be changed. Finally, in Section 7.5 we show how to change the data representation in program descriptions.

The soundness of the stronger proof rules will be shown by deriving them from the general proof rule for refinement. The derivations will essentially be carried out in $L_{\omega_1\omega}$, using the axioms and inference rules of this logic. One of the main purposes of this chapter is in fact to illustrate the power of the general proof rule for refinement and the suitability of $L_{\omega_1\omega}$ as a formal system in which to reason about program properties.

Refinements will be restricted to be of the form $S \leq S'$, where S is a program description and S' is either a program description or an abstraction of a program description. The proof rules to be presented here can therefore all be used in the safe development of programs. The proof rules are not intended to form in any sense a complete set of rules for deriving programs, but are

mainly given as examples if important kinds of rules which can be expressed and proved correct in our framework.

7.1. AN EXAMPLE OF FORMAL PROGRAM DEVELOPMENT

We will here show how the example of Section 6.1 can be formally developed using program descriptions and the principles laid down in Section 6.2.

The problem specification can be expressed as the program description

$$A_0\colon \underline{\text{if}}\ X > 1 \wedge Y \geq 0 \rightarrow z := X^Y\ \underline{\text{fi}}\colon V \rightarrow V,$$

where $V = \{X,Y,z\}$. Thus the problem is to construct a program description S such that $A_0 \leq S$. The solution S is constrained by requiring that the exponentiation operation is not used. The variable sets (like V above) will be omitted in the sequel.

We will introduce the abbreviation

$$R_1\colon X > 1 \wedge Y \geq 0$$

for future convenience. Thus A_0 is

$$A_0\colon \underline{\text{if}}\ R_1 \rightarrow z := X^Y\ \underline{\text{fi}}.$$

We will assume that the variables take only integers as values. This means that we postulate a set Δ of sentences (the axioms) which give the operations used in the program descriptions the properties expected of the usual integer operations. This set Δ will not be mentioned explicitly in the example below, ($S \leq S'$ is to be understood as stating that $S \leq S'$ is a logical consequence of Δ).

As the first refinement step we introduce some assertions into A_0. Let A_1 be

$$A_1\colon \underline{\text{if}}\ R_1 \rightarrow \{R_1\};\ z := X^Y\ \underline{\text{fi}}.$$

The fact that $A_0 \leq A_1$ holds follows by a transformation rule for introducing assertions (the rule is given in example 7.7(i), Section 7.3).

We now try to find a refinement S' of the specification

$$B_0\colon\ \{R_1\};\ z := X^Y.$$

If we find such a refinement, i.e. an S' satisfying

$$B_0 \leq S',$$

then the replacement theorem implies that

$$A_0 \leq A_1 \leq \underline{\text{if}}\ R_1 \rightarrow S'\ \underline{\text{fi}},$$

thus giving us the required solution.

The following is a refinement of B_0:

B_1: $\{R_1\}$;
  beg h:
    h,z := X^Y, 1; $\{R_2\}$;
    do h $\neq$ 1 $\rightarrow$ h,z := h',z'.($1 \leq h' < h \wedge R_2'$);
      $\{R_2\}$
    od
  end

We use the abbreviations

$$R_2\colon\ h \cdot z = X^Y \wedge h \geq 1$$

and

$$R_2'\colon\ h' \cdot z' = X^Y \wedge h' \geq 1.$$

The assignment

$$h,z := h',z'.(1 \leq h' < h \wedge R_2')$$

is obviously finite on integers, and has the effect described in Section 6.1 by

"squeeze h under invariance of P".

The way to prove that $B_0 \leq B_1$ holds is given in example 7.1, Section 7.2.

The invariant R_2 in B_1 is a byproduct we get when showing the correctness of the implementation by the invariant technique, loosely described in Section 6.1 and more thoroughly treated in DIJKSTRA [15]. It comes in very handy when preparing for a replacement in context.

Our next step is to get rid of the abstract variable h using the variables x and y to represent the value of h. This constituted the second step in Section 6.1. It will, however, take us more than one refinement step to make this passage.

We prepare for this step by collecting some necessary information in the form of assertions in the program description. This gives us the refinement B_2 of B_1:

```
B2: {R1};
    beg h:
          {R1}; h,z := X^Y, 1; {R2};
          do h ≠ 1 → {R2 ∧ h ≠ 1};
                     h,z := h',z'.(1 ≤ h' < h ∧ R2'); {R2}
          od
    end.
```

The fact that $B_1 \leq B_2$ holds can be shown by using the appropriate transformation rules for introducing assertions into program descriptions. We would need the transformation rules of example 7.8 and 7.9(v) to get from B_1 to B_2.

We will now consider the following two components of B_2:

$$C_0\colon \{R_1\};\ h,z := X^Y,\ 1$$

and

$$D_0\colon \{R_2 \wedge h \neq 1\};\ h,z := h',z'.(1 \leq h' < h \wedge R_2').$$

The program description C_0 will be implemented with the description

```
C1: beg x,y/h.Q:
        x,y,z := X,Y,1
    per
```

where Q is

Q: $h = x^y \wedge x > 1$.

The effect of C_1 is to initialize the variables h and z to X^Y and 1, as required, by first computing appropriate values for x, y and z, and then assigning to h the value represented by x and y. The form beg ... per is used here, because the initial value of h is not needed to compute the value required. The way in which $C_0 \leq C_1$ is to be proved is discussed in example 7.2 of Section 7.2.

The program description D_0 will again be implemented with the description

D_1: rep x,y/h.Q:
    y,z := y-1,z·x
  per .

Because the initial value of h is referred to in D_0, we use the form rep ... per. The way in which $D_0 \leq D_1$ is to be proved is discussed in example 7.3 of Section 7.2.

The proof rules for abstraction (Section 7.5) can now be used to change the data representation in B_2. From $C_0 \leq C_1$ and $D_0 \leq D_1$, together with the fact that $h \neq 1$ iff $y \neq 0$ when Q holds, we first get that

do $h \neq 1 \rightarrow D_0$ od $\leq$ rep x,y/h.Q:
    do $y \neq 0 \rightarrow D_1$ od
  per.

This together with $C_0 \leq C_1$ can then be used to get

C_0; do $h \neq 1 \rightarrow D_0$ od $\leq$ beg x,y/h.Q:
    C_1; do $y \neq 0 \rightarrow D_1$ od
  per,

From this we finally get

```
beg h: C0;                          beg x,y: C1;
       do h ≠ 1 → D0 od      ≤             do y ≠ 0 → D1 od
end                                 end
```

Substituting the right hand side for the left hand side in B_2 will now give us a refinement of B_0. The component B_0 of A_1 can therefore be replaced with this, giving solution A_2 below (this corresponds to step S_2 in Section 6.1):

```
A2: if X > 1 ∧ Y ≥ 0 →
        beg x,y:
                x,y,z := X,Y,1;
                do y ≠ 0 → y,z := y-1,z•x od
        end
    fi
```

To get step S_3 in Section 6.1, we backtrack to the program description B_2, and give the refinement B_3' of it instead:

```
B3': {R1};
    beg h:
          {R1}; h,z := X^Y, 1; {R2};
          do h ≠ 1 → {R2 ∧ h ≠ 1}; skip;
                     {R2 ∧ h ≠ 1};
                     h,z := h',z'.(1 ≤ h' < h ∧ R2'); {R2}
          od
    end
```

It is quite obvious that $B_3 \leq B_3'$, as the skip statement does not affect the values of any program variables. We then consider the components C_0 and D_0 of B_3', which are the same as the components C_0 and D_0 of B_3, and implement these as before with C_1 and D_1. We will also consider the component

```
E0: {R2 ∧ h ≠ 1}; skip
```

of B_3'. This component will be implemented by

```
E1: rep x,y/h.Q:
        do 2|y → x,y := x•x,y/2 od
    per.
```

The way in which $E_0 \leq E_1$ is to be proved is shown in example 7.4 of Section 7.2.

We then proceed to change the data representation in B_3', in a similar manner as above. After having done this, we get the program description

```
B4': beg x,y:
        x,y,z := X,Y,1;
        do y ≠ 0 → do 2|y → x,y := x•x,y/2 od;
                   y,z := y-1,z•x
        od
    end,
```

where $B_3' \leq B_4'$. By replacing B_0 in A_1 with B_4', we then get the program description A_2', which corresponds to the step S_3 in Section 6.1:

```
A2': if X > 1 ∧ Y ≥ 0 →
         beg x,y:
             x,y,z := X,Y,1;
             do y ≠ 0 → do 2|y → x,y := x•x,y/2 od;
                        y,z := y-1,z•x
             od
         end
    fi.
```

We now subject our program to a last refinement. It does not make the program more efficient, on the contrary, but it makes it structurally simpler, by fusing the two nested loops of the program into one single loop. This transformation is not done by Dijkstra, for obvious reasons. Our purpose here is to show how the control structure of programs can be changed.

We consider the following component F_0 in A_2':

```
F0: do y ≠ 0 → do 2|y → x,y := x•x,y/2 od;
               y,z := y-1,z•x
    od.
```

Using a program transformation on loops, to be proved correct in example 7.10 of Section 7.4, we get the refinement F_1 of F_0:

F_1: do y ≠ 0 → if 2|y → x,y := x•x,y/2
□ ~2|y → y,z := y-1,z•x fi
od

The program description F_1 will be less efficient than F_0 because the condition y≠0 is tested at each iteration, wheras this test is not performed in F_0 while iterating in the inner loop.

Replacing F_0 by F_1 in A_2' gives us the solution A_3' to the programming problem, where A_3' is

A_3': if X > 1 ∧ Y ≥ 0 →
beg x,y:
x,y,z := X,Y,1;
do y ≠ 0 → if 2|y → x,y := x•x,y/2
□ ~2|y → y,z := y-1,z•x fi
od
end
fi

7.2. PROOF RULES FOR IMPLEMENTATION

We will give here a general proof rule by which the correctness of an *implementation*, i.e. of a refinement of the form

{P}; x := x'.Q ≤ S

can be shown. For this purpose, we need to prove a technical lemma first.

LEMMA 7.1. *For any set Δ of sentences, we have*

(7.1) $\Delta \vdash \forall x'(Q \Rightarrow R[x'/x])$

iff

(7.2) $\Delta \vdash \forall x_0 y_0 (x=x_0 \wedge y=y_0 \Rightarrow \forall xy(Q[x_0/x, x/x'] \wedge y=y_0 \Rightarrow R))$,

when var(Q) $\subseteq$ V $\cup$ $\tilde{x}'$ *and* var(R) $\subseteq$ V, $\tilde{x}_0$ *and* $\tilde{y}_0$ *do not contain variables of* V *or* $\tilde{x}$ *or* $\tilde{y}$, $\tilde{x} \cap \tilde{y} = \emptyset$ *and* $\tilde{x}_0 \cap \tilde{y}_0 = \emptyset$ *and* $\tilde{y} \subseteq$ V.

PROOF. By Lemma 2.6, (7.2) is equivalent to

$$(\forall xy(Q[x_0/x,x/x'] \wedge y=y_0 \Rightarrow R))[x/x_0,y/y_0].$$

By changing the bound variables x and y, this gives us

$$(\forall x'y'(Q[x_0/x,y'/y] \wedge y'=y_0 \Rightarrow R[x'/x,y'/y]))[x/x_0,y/y_0],$$

thus making x_0 and y_0 free for x and y. Because x_0 and y_0 do not occur free in R[x'/x,y'/y] and y_0 does not occur free in $Q[x_0/x,y'/y]$, performing the substitution gives us the result

$$\forall x'y'(Q[y'/y] \wedge y'=y \Rightarrow R[x'/x,y'/y]).$$

This is again equivalent to

$$\forall x'y'(y=y' \Rightarrow (Q[y'/y] \Rightarrow R[x'/x,y'/y])),$$

giving the equivalent form

$$\forall x'(Q \Rightarrow R[x'/x]),$$

by using Lemma 2.6 again. This is the desired result, so the lemma is proved. □

The general proof rule for establishing the correctness of an implementation is now given by the following theorem.

THEOREM 7.2. *Let* Δ *be a countable set of sentences of* L. *Let* V *be a finite nonempty set of program variables, and let* S *be a program description or an abstraction in* V. *Let* y *be a list of those variables in* V - $\tilde{x}$ *that are not constant in* S. *Let* x_0 *and* y_0 *be lists of distinct program variables not occurring in* S *or belonging to* V. *If*

$$\Delta \vdash P \wedge x=x_0 \wedge y=y_0 \Rightarrow WP(S,Q[x_0/x,x/x'] \wedge y=y_0),$$

then

$$\Delta \vdash \{P\};\ x := x'.Q \leq S.$$

PROOF. Let k be the number of variables in V, and let v be a list of distinct variables, $\tilde{v} = V$. Let G be a new k-place predicate symbol. By Cor. 5.5, it is sufficient to show that

$$\Delta \vdash WP(\{P\};\ x := x'.Q,\ G(v)) \Rightarrow WP(S,G(v)).$$

Take therefore WP({P}; x := x'.Q,G(v)) as an assumption, i.e. we assume that

(7.3) $$P \wedge \exists x'.Q \wedge \forall x'(Q \Rightarrow G(v)[x'/x]).$$

Note that the assumption may contain free variables of V, over which we are not allowed to quantify. By Lemma 7.1, the third term in the assumption implies that we have

$$\forall x_0 y_0 (x=x_0 \wedge y=y_0 \Rightarrow \forall xy(Q[x_0/x,x/x'] \wedge y=y_0 \Rightarrow G(v))).$$

Using axiom Q2, this gives us that

$$x=x_0 \wedge y=y_0 \Rightarrow \forall xy(Q[x_0/x,x/x'] \wedge y=y_0 \Rightarrow G(v)).$$

Let us now further assume that

(7.4) $$x=x_0 \wedge y=y_0.$$

By modus ponens we get that

$$\forall xy(Q[x_0/x,x/x'] \wedge y=y_0 \Rightarrow G(v)).$$

Because all variables of V not belonging to x or y are constant in S, we may apply Lemma 5.11(i), getting the result

$$WP(S,Q[x_0/x,x/x'] \wedge y=y_0) \Rightarrow WP(S,G(v)).$$

Because of the assumptions and the premise, we have that

$$WP(S,Q[x_0/x,x/x'] \wedge y=y_0),$$

and thus we may infer by modus ponens that

$$WP(S,G(v)).$$

We still have to get rid of the assumptions that we made in the course of developing the proof. By the deduction theorem, we first get that

$$x=x_0 \wedge y=y_0 \Rightarrow WP(S,G(v)),$$

thus getting rid of assumption (7.4). As x_0 and y_0 are not free in assumption (7.3), we may use the rule GN on this, getting

$$\forall x_0 y_0 (x=x_0 \wedge y=y_0 \Rightarrow WP(S,G(v)),$$

which then gives us

$$WP(S,G(v))[x/x_0,y/y_0]$$

by Lemma 2.6 i.e.

$$WP(S,G(v)),$$

by noting that x_0 and y_0 are not free in WP(S,G(v)) (Lemma 5.1). Using the deduction theoren once again, we eliminate assumption (7.3), getting the desired result

$$WP(\{P\};\ x := x'.Q,\ G(v)) \Rightarrow WP(S,G(v)). \quad \square$$

COROLLARY 7.3. *Let the assumptions be as in Theorem* 7.2. *We then have that*

$$\vdash \exists x'.Q \wedge x=x_0 \wedge y=y_0 \Rightarrow WP(S,Q[x_0/x,x/x'] \wedge y=y_0))$$

implies

$$\vdash x := x'.Q \leq S.$$

PROOF. By noting that x := x'.Q ≈ {∃x'.Q}; x := x'.Q. □

COROLLARY 7.4. *Let the assumption be as in Theorem 7.2. Then for the assignment statement* x := t *in* V, *we have that*

$$\vdash P \wedge x=x_0 \wedge y=y_0 \Rightarrow WP(S,x=t[x_0/x] \wedge y=y_0)$$

implies

$$\vdash \{P\};\ x := t \leq S.$$

PROOF. Immediate. □

We will now show how the implementation steps in Section 7.1 can be proved correct, using the proof rules for implementations.

EXAMPLE 7.1. The first implementation step was the refinement of B_0 to B_1. Thus we have to prove that $B_0 \leq B_1$, where

$$B_0\colon \{X > 1 \wedge Y \geq 0\};\ z := X^Y.$$

We apply here Corollary 7.4. Using the notation of this corollary, we have in this case that y = <>, because B_1 only affects the variable z. Also, the assignment performed is an initialization, i.e. the variable x does not occur in t (here: the variable z does not occur in X^Y). In this case, the premise in Corollary 7.4 simplifies to

$$P \Rightarrow WP(S,x=t).$$

Thus we have to prove that

$$X > 1 \wedge Y \geq 0 \Rightarrow WP(B_1,z=X^Y).$$

We will not prove this here. An informal argument was given in Section 6.1. A more formal proof can also be given, based on the "fundamental invariance theorem" in DIJKSTRA [15].

EXAMPLE 7.2. The second implementation was the implementation of C_0 with C_1, where

$$C_0\colon \{X > 1 \wedge Y \geq 0\};\ h,z := X^Y,\ 1$$

and

$$C_1\colon \underline{\text{beg}}\ x,y/h.(h=x^Y \wedge x > 1)\colon\ x,y,z := X,Y,1\ \underline{\text{per}}.$$

This is again an initializing assignment not affecting variables other than those indicated, so we can use the same proof rule as in example 7.1. Thus we have to prove that

$$X > 1 \wedge Y \geq 0 \Rightarrow WP(C_1,\ h=X^Y \wedge z=1).$$

The weakest precondition for C_1 can be calculated using the formula for weakest preconditions of abstraction in Section 6.3, which is

$$WP(\underline{\text{beg}}\ x/y.y=t \wedge I\colon\ S\ \underline{\text{per}},\ R) \Leftrightarrow \forall x WP(S, I \wedge R[t/y]).$$

(Here x and y are variable lists, and should not be confused with x and y in the example.) Using it in the present example means that we have to prove

$$X > 1 \wedge Y \geq 0 \Rightarrow \forall xy WP(x,y,z := X,Y,1,(x > 1 \wedge x^Y = X^Y \wedge z=1)).$$

Using the rule for computing the weakest precondition of an assignment statement, also given in Section 6.3, the premise to be proved becomes

$$X > 1 \wedge Y \geq 0 \Rightarrow \forall xy(X > 1 \wedge X^Y = X^Y \wedge 1 = 1),$$

i.e., we have to prove that

$$X > 1 \wedge Y \geq 0 \Rightarrow X > 1 \wedge X^Y = X^Y \wedge 1=1,$$

which obviously holds. Thus we conclude that $C_0 \leq C_1$.

EXAMPLE 7.3. The third implementation was the implementation of D_0 with D_1, where

$$D_0\colon \{R_2 \wedge h \neq 1\};\ h,z := h',z'.(1 \leq h' < h \wedge R_2'),$$

and

$$D_1\colon \underline{\text{rep}}\ x,y/h.(h = x^y \wedge x > 1)\colon\ y,z := y-1, z\cdot x\ \underline{\text{per}}$$

Here

$$R_2\colon\ h\cdot z = X^Y \wedge h \geq 1$$

and

$$R_2'\colon\ h'\cdot z' = X^Y \wedge h' \geq 1.$$

We use Theorem 7.2 here. Because y = <>, the premise in 7.2 takes the form

$$P \wedge x = x_0 \Rightarrow WP(S, Q[x_0/x, x/x']).$$

Thus in the present case, we have to prove that

$$R_2 \wedge h \neq 1 \wedge h = h_0 \wedge z = z_0 \Rightarrow WP(D_1,\ 1 \leq h < h_0 \wedge R_2).$$

The weakest precondition for the abstraction D_1 is given by the formula

$$\begin{aligned} &WP(\underline{\text{rep}}\ x/y.y=t \wedge I\colon\ S\ \underline{\text{per}},\ R) \\ &\qquad \Longleftrightarrow \exists x(y=t \wedge I) \wedge \\ &\qquad\quad \forall x(y=t \wedge I \Rightarrow WP(S, I \wedge R[t/x])). \end{aligned}$$

Thus, in order to establish that $D_0 \leq D_1$, we have to prove here that

$$\begin{aligned} &R_2 \wedge h \neq 1 \wedge h = h_0 \wedge z = z_0 \\ &\Rightarrow \exists xy(h = x^y \wedge x > 1) \\ &\wedge \forall xy(h = x^y \wedge x > 1 \Rightarrow WP(y,z := y-1, z\cdot x, (x > 1 \wedge 1 \leq x^y < h_0 \wedge R_2[x^y/h]))). \end{aligned}$$

The first term of the conjunction is clearly implied by the left hand side, by taking x = h, y = 1. This together with some other simplifications gives us the formula

$$\begin{aligned} &h\cdot z = X^Y \wedge h > 1 \wedge h = h_0 \wedge h = x^y \wedge x > 1 \\ &\Rightarrow x > 1 \wedge 1 \leq x^{y-1} < h_0 \wedge x^{y-1}\cdot z\cdot x = X^Y \wedge x^{y-1} \geq 1. \end{aligned}$$

Using the properties of integer arithmetic, this formula can be seen to hold.

EXAMPLE 7.4. The last implementation that we performed in Section 7.1 was the implementation of E_0 with E_1, where

$$E_0\colon \{R_2 \wedge h\neq 1\};\ \mathbf{skip}$$

and

$$E_1\colon \underline{\mathbf{rep}}\ x,y/h.(h=x^y \wedge x>1):\ \underline{\mathbf{do}}\ 2|y \rightarrow x,y := x\cdot x, y/2\ \underline{\mathbf{od}}\ \underline{\mathbf{per}}.$$

To prove that $E_0 \leq E_1$, we can still use the Theorem 7.2, because

$$\mathbf{skip} \approx \langle\rangle := \langle\rangle.\mathbf{true}.$$

In this case the premise of Theorem 7.2 takes the form

$$P \wedge y=y_0 \Rightarrow WP(S, y=y_0),$$

because Q = true. Thus we have to prove that

$$R_2 \wedge h\neq 1 \wedge h=h_0 \Rightarrow WP(E_1, h=h_0).$$

Computing the weakest precondition gives us the formula

$$R_2 \wedge h\neq 1 \wedge h=h_0 \Rightarrow \forall xy(h=x^y \wedge x>1 \Rightarrow WP(E_1', x>1 \wedge x^y=h_0)),$$

where

$$E_1'\colon \underline{\mathbf{do}}\ 2|y \rightarrow x,y := x\cdot x, y/2\ \underline{\mathbf{od}},$$

where we omitted the conjunct $\exists xy(h=x^y \wedge x>1)$ because it was already proved to follow from the assumptions given (in example 7.3). This can be proved by the usual invariant technique, referred to in example 7.1, by taking the condition

$$x>1 \wedge x^y=h_0 \wedge h_0>1$$

as the loop invariant. The loop will terminate because each turn around the loop will decrease the number of factors 2 in y, while $\sim 2 \mid y$ will hold if and only if the number of factors 2 in y is zero.

7.3. TRANSFORMATION RULES FOR ASSERTIONS

As shown in Chapter 6.2, assertions play an important part in program development by stepwise refinement, as formalized in this tract. Therefore, proof rules are needed by which assertions can be introduced at various places in program descriptions. The assertions introduced give information about the context in which they appear, thereby making it easier to find a correct replacement for a component.

We will not present a complete list of assertion rules to be used in program development, but will restrict ourselves to only give examples of such rules. The examples are partly chosen to show the correctness of the refinement steps made in Section 7.1 and partly for later use.

Before going into the examples, we will, however, prove another form of the result in Lemma 5.10(i), which gave the induction rule for loops.

LEMMA 7.5. *Let* Δ *be a countable set of sentences of* L. *If*

$$\Delta \vdash \{P\};\ \underline{\text{do}}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{od}}^n \leq S, \quad \text{for } n < \omega,$$

then

$$\Delta \vdash \{P\};\ \underline{\text{do}}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{od}} \leq S.$$

PROOF. Let the program descriptions above all be program descriptions in V, where V is a finite nonempty set of program variables. Let $\tilde{v} = V$ and G be as usual. Using the abbreviations

$$DO^n = \underline{\text{do}}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{od}}^n$$

and

$$DO = \underline{\text{do}}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{\text{od}},$$

we have to prove that

$$\text{(7.4)} \qquad WP(\{P\};\ DO^n, G(v)) \Rightarrow WP(S, G(v)) \quad \text{for } n < \omega$$

implies

(7.5) $\quad WP(\{P\};\ DO,G(v)) \Rightarrow WP(S,G(v)).$

The assumption (7.4) gives us that

$$P \wedge WP(DO^n,G(v)) \Rightarrow WP(S,G(v)), \quad \text{for } n < \omega,$$

or equivalently

$$P \Rightarrow (WP(DO^n,G(v)) \Rightarrow WP(S,G(v))), \quad \text{for } n < \omega.$$

Make the assumption P. We then have that

$$WP(DO^n,G(v)) \Rightarrow WP(S,G(v)), \quad \text{for } n < \omega,$$

and using the inference rule for infinite disjunction, we get that

$$\bigvee_{n<\omega} WP(DO^n,G(v)) \Rightarrow WP(S,G(v)),$$

i.e.

$$WP(DO,G(v)) \Rightarrow WP(S,G(v)).$$

Using the deduction theorem, we get from this that

$$P \Rightarrow (WP(DO,G(v)) \Rightarrow WP(S,G(v))),$$

i.e.

$$P \wedge WP(DO,G(v)) \Rightarrow WP(S,G(v)),$$

which gives the final result (7.5). □

EXAMPLE 7.5. If $\Delta \vdash P \Rightarrow P'$ then $\Delta \vdash \{P\} \leq \{P'\}$. This is obvious, by considering

$$WP(\{P\},G(v)) \Rightarrow WP(\{P'\},G(v)),$$

i.e.

$$P \wedge G(v) \Rightarrow P' \wedge G(v).$$

Because $\Delta \vdash P \Rightarrow$ true for any P, we have $\Delta \vdash \{P\} \leq$ skip for any P, remembering that skip = {true}. Therefore, an assertion may always be replaced with the skip statement, and the resulting description will be a refinement of the original description. Thus we are always allowed to remove an assertion without affecting the correctness of the program description.

EXAMPLE 7.6. If $\Delta \vdash P \Rightarrow WP(S,Q)$, then $\Delta \vdash \{P\}; S \leq \{P\}; S; \{Q\}$. This is also easily seen, because

$$WP(\{P\}; S,G(v)) \Longleftrightarrow P \wedge WP(S,G(v))$$

and

$$\begin{aligned} P \wedge WP(S,G(v)) &\Rightarrow P \wedge WP(S,Q) \wedge WP(S,G(v)) \\ &\Rightarrow P \wedge WP(S,Q \wedge G(v)) \qquad \text{(Lemma 5.11(i))} \\ &\Leftrightarrow WP(\{P\}; S; \{Q\},G(v)). \end{aligned}$$

Thus, using the previous example, we have that $\Delta \vdash P \Rightarrow WP(S,Q)$ implies that $\Delta \vdash \{P\};S \approx \{P\};S;\{Q\}$.

EXAMPLE 7.7. The facts that

(i) $$\begin{aligned} &\Delta \vdash \{P\}; \underline{\text{if}}\ B_1 \rightarrow S_1 \,\square \ldots \square\, B_n \rightarrow S_n\ \underline{\text{fi}} \\ &\quad \approx \{P\}; \underline{\text{if}}\ B_1 \rightarrow \{P \wedge B_1\}; S_1 \,\square \ldots \square\, B_n \rightarrow \{P \wedge B_n\}; S_n\ \underline{\text{fi}} \end{aligned}$$

and

(ii) $$\begin{aligned} &\Delta \vdash \underline{\text{if}}\ B_1 \rightarrow S_1 \,\square \ldots \square\, B_n \rightarrow S_n\ \underline{\text{fi}}; \{Q\} \\ &\quad \approx \underline{\text{if}}\ B_1 \rightarrow S_1; \{Q\} \,\square \ldots \square\, B_n \rightarrow S_n; \{Q\}\ \underline{\text{fi}} \end{aligned}$$

also follow directly, by analyzing the corresponding weakest preconditions.

EXAMPLE 7.8. We use the previous examples and Lemma 7.5 to show that

$$\begin{aligned} &\Delta \vdash \{P\}; \underline{\text{do}}\ B_1 \rightarrow S_1;\{P\} \,\square \ldots \square\, B_n \rightarrow S_n;\{P\}\ \underline{\text{od}} \\ &\quad \approx \{P\}; \underline{\text{do}}\ B_1 \rightarrow \{B_1 \wedge P\};S_1;\{P\} \,\square \ldots \square\, B_n \rightarrow \{B_n \wedge P\};S_n;\{P\}\ \underline{\text{od}}. \end{aligned}$$

Denote the left hand side by {P};DO and the right hnad side by {P};DO'. By example 7.5, we only need to show that {P};DO ≤ {P};DO'. Using the Lemma 7.5, to show this, it is sufficient to show that

$$\{P\};DO^n \leq \{P\};DO', \quad \text{for } n < \omega.$$

Because $DO'^n \leq DO'$ for every $n < \omega$, it will be sufficient to show that

$$\{P\};DO^n \leq \{P\};DO'^n \quad \text{for } n < \omega.$$

We prove it by induction on n. For n = 0 this result is obvious, as $\{P\};DO^0 \approx$ abort. Assume that the refinement holds for n, $n < \omega$.

By the definition of DO^n, we then have that

$$\{P\};DO^{n+1}$$

$$= \{P\};\ \underline{\text{if}}\ BB \to \underline{\text{if}}\ B_1 \to S_1;\{P\} \,\square \ldots \square\, B_n \to S_n;\{P\}\ \underline{\text{fi}};\ DO^n$$
$$\qquad \square\ \sim BB \to \text{skip}\ \underline{\text{fi}}$$

$$\leq \{P\};\ \underline{\text{if}}\ BB \to \{P\};\ \underline{\text{if}}\ B_1 \to S_1;\{P\} \,\square \ldots \square\, B_n \to S_n;\{P\}\ \underline{\text{fi}};\ DO^n$$
$$\qquad \square\ \sim BB \to \text{skip}\ \underline{\text{fi}}$$

$$\leq \{P\};\ \underline{\text{if}}\ BB \to \underline{\text{if}}\ B_1 \to \{P \wedge B_1\};S_1;\{P\}\ldots$$
$$\qquad\qquad \square\ B_n \to \{P \wedge B_n\};S_n;\{P\}\ \underline{\text{fi}};\ \{P\};DO^n$$
$$\qquad \square\ \sim BB \to \text{skip}\ \underline{\text{fi}}$$

$$\leq \{P\};DO'^{n+1}.$$

In the first refinement above, we used example 7.7(i) and 7.5 (the latter e.g. when replacing P ∧ BB with P, because P ∧ BB ⇒ P). In the second refinement we used example 7.7(i) and (ii), as well as example 7.5. The last refinement used the induction hypothesis, and the definition of DO'^n.

EXAMPLE 7.9. Finally we have an assertion rule concerned with blocks (the soundness of this rule can be checked by considering the corresponding weakest preconditions, as was done in the preceding examples):

$$\Delta \vdash \{P\};\ \underline{\text{beg}}\ x\colon S\ \underline{\text{end}};\ \{Q\}$$

$$\approx \{P\};\ \underline{\text{beg}}\ x\colon \{P\};S;\{Q\}\ \underline{\text{end}};\ \{Q\}.$$

In the refinements of Section 7.1, rules for assertions were needed in the refinements of A_0 to A_1 and of B_1 to B_2. In the first case, the fact that $A_0 \leq A_1$ can be justified using the rule in example 7.7(i), while the fact that $B_1 \leq B_2$ holds can be justified using the rules in example 7.8 and example 7.9.

7.4. TRANSFORMATION RULES FOR CONTROL STRUCTURES

We next outline the technique for showing the correctness of program transformations involving control structures. We will not be as formal here as in the preceding chapters, and feel free to use some obvious, but unproven results. We use the refinement of F_0 to F_1 in the example of Section 7.1 to illustrate the technique.

The refinement of F_0 to F_1 can be justified by the following rule for loops.

EXAMPLE 7.10. Let

$$DO = \underline{\text{do}}\ B \to DO';S\ \underline{\text{od}},$$

$$DO' = \underline{\text{do}}\ B' \to S'\ \underline{\text{od}}$$

and

$$DO'' = \underline{\text{do}}\ B \to \underline{\text{if}}\ B' \to S' \ \square\ \sim B' \to S\ \underline{\text{fi}}\ \underline{\text{od}}.$$

Assume that $\{B \wedge B'\};S' \leq \{B \wedge B'\};S';\{B\}$. Then $DO \leq DO''$.

We first show that

(7.7) $\quad \{B\};DO'^{n};S;DO \leq \{B\};DO'', \quad$ for $n < \omega$.

For $n = 0$ this is obvious, as $\{B\};DO'^{0};S;DO \approx$ abort. Assume that (7.7) holds for n, $n < \omega$. We have that

$$\{B\};DO'^{n+1};S;DO = \{B\};\ \underline{\text{if}}\ B' \to S';DO'^{n} \ \square\ \sim B' \to \text{skip}\ \underline{\text{fi}};\ S;DO.$$

Consider now separately the two cases $B \wedge B'$ and $B \wedge \sim B'$.

(i) $B \wedge B'$. We have that

$$\{B \wedge B'\};DO'^{n+1};S;DO \leq \{B \wedge B'\};S';DO'^{n};S;DO$$

$$\leq \{B \wedge B'\};S';\{B\};DO'^{n};S;DO$$

because the alternative B' must in this case be chosen in DO'^{n+1}. Using the induction hypothesis, this gives us that

$$\{B \wedge B'\};DO'^{n+1};S;DO \leq \{B \wedge B'\};S';DO".$$

Because of the condition $B \wedge B'$, we have

$$\{B \wedge B'\};S';DO" \leq \{B \wedge B'\};\ \underline{\text{if}}\ B \rightarrow \underline{\text{if}}\ B' \rightarrow S';DO"$$
$$\square\ \sim B' \rightarrow S;DO"\ \underline{\text{fi}}$$
$$\square\ \sim B \rightarrow \text{skip}\ \underline{\text{fi}}$$

$$\leq \{B \wedge B'\};\ \underline{\text{if}}\ B \rightarrow \underline{\text{if}}\ B' \rightarrow S' \square\ \sim B' \rightarrow S\ \underline{\text{fi}};$$
$$DO"$$
$$\square\ \sim B \rightarrow \text{skip}\ \underline{\text{fi}}$$

$$\leq \{B \wedge B'\};DO".$$

Thus we have that

$$\{B \wedge B'\};DO'^{n+1};S;DO \leq \{B \wedge B'\};DO".$$

(ii) $B \wedge \sim B'$. We have that

$$\{B \wedge \sim B'\};DO'^{n+1};S;DO \leq \{B \wedge \sim B'\};S;DO,$$

as the loop will not be entered when B' is false. For the same reason, we have that

$$\{B \wedge \sim B'\};S;DO \leq \{B \wedge \sim B'\};\{B\};DO'^{n};S;DO$$

$$\leq \{B \wedge \sim B'\};DO",$$

by use of the induction hypothesis. Thus we have that

$$\{B \wedge \sim B'\};DO'^{n+1};S;DO \leq \{B \wedge \sim B'\};DO".$$

Putting these two cases together gives the required result, i.e. we get that

$$\{B\};DO'^{n+1};S;DO \leq \{B\};DO",$$

which proves that (7.12) holds for every $n < \omega$. From this we infer that

$$\{B\};DO';S;DO \leq \{B\};DO".$$

This inference can be proved correct with a similar argument as was used in the proof of Lemma 7.5.

We now turn to our main task, i.e. to proving that DO $\leq$ DO". We show this by showing that

(7.8) $\quad DO^n \leq DO", \quad$ for $n < \omega$.

For $n = 0$ this is immediate, as usual. Assume that (7.8) holds for n, $n < \omega$. We then have that

$$DO^{n+1} = \underline{\text{if}}\ B \to DO';S;DO^n \ [] \ \sim B \to \text{skip}\ \underline{\text{fi}}$$

$$\leq \underline{\text{if}}\ B \to \{B\};DO';S;DO \ [] \ \sim B \to \text{skip}\ \underline{\text{fi}}$$

$$\leq \underline{\text{if}}\ B \to \{B\};DO" \ [] \ \sim B \to \text{skip}\ \underline{\text{fi}}$$

$$\leq \underline{\text{if}}\ B \to \underline{\text{if}}\ B' \to S' \ [] \ \sim B' \to S\ \underline{\text{fi}};\ DO"$$
$$[] \ \sim B \to \text{skip}\ \underline{\text{fi}}$$

$$\leq DO".$$

In these steps we have made use of the fact that

$$DO" \approx \underline{\text{if}}\ B \to \underline{\text{if}}\ B' \to S' [] \ \sim B' \to S\ \underline{\text{fi}};\ DO"$$
$$[] \ \sim B \to \text{skip}\ \underline{\text{fi}}.$$

The derivation shows that (7.8) holds, thus proving the desired results.

7.5. TRANSFORMATION RULES FOR ABSTRACTIONS

In this final section we give some rules for handling abstractions in refinements. The purpose of these rules is to enable one to change the data representation in a program. As in the previous sections of this chapter, we do not aim at a complete set of rules, but will be content with giving the most basic ones, mainly in order to show the correctness of the program transformation rules used in developing the example program in Section 7.1. The rules given will also not always be in the most general form possible.

For the formulation of the lemmas below, let us fix a countable set Δ of sentences of L. Let v and w be two sets of program variables, let $S_i'\colon W \to W$ be program descriptions, i = 1,...,n, and let

(7.9) $$S_i = \underline{\text{rep}}\ \beta\colon S_i'\ \underline{\text{per}},$$

for i = 1,...,n, where

$$\beta = x/y.(y=t \wedge I),$$

$\text{var}(t) \subseteq W$ and $\text{var}(I) \subseteq W$.

LEMMA 7.6. *If* $\Delta \vdash P \Rightarrow \exists x(y=t \wedge I)$, *then*

$$\Delta \vdash \{P\};\ \text{skip} \leq \underline{\text{rep}}\ \beta : \text{skip}\ \underline{\text{per}}.$$

PROOF. The case here is similar to the case in example 7.4, and we have to prove that

(7.10) $$P \wedge y=y_0 \Rightarrow WP(\underline{\text{rep}}\ \beta : \text{skip}\ \underline{\text{per}},\ y=y_0).$$

Computing the weakest precondition, this gives us the formula

$$P \wedge y=y_0 \Rightarrow \forall x(y=t \wedge I \Rightarrow t=y_0 \wedge I)$$

where we have used the assumption that $P \Rightarrow \exists x(y=t \wedge I)$ to eliminate the formula $\exists x(y=t \wedge I)$ on the right hand side of (7.10).

Let us assume that

$$P \wedge y=y_0 \wedge y=t \wedge I.$$

This gives the result that

$$t=y_0 \wedge I,$$

and by the deduction theorem, we have that

$$y=t \wedge I \Rightarrow t=y_0 \wedge I,$$

under the assumption $P \wedge y=y_0$. As x is not free in this assumption, we get

$$\forall x(y=t \wedge I \Rightarrow t=y_0 \wedge I),$$

and another application of the deduction theorem will then give the desired result. □

LEMMA 7.7. *Assume that* S_i *and* S_i' *are as in* (7.9), i = 1,...,n. *Then*

$$\Delta \vdash S_1;\ldots;S_n \leq \underline{\text{rep}}\ \beta : S_1';\ldots;S_n' \ \underline{\text{per}}.$$

PROOF. We prove the case for n = 2, the general case follows by induction on n. For the proof, let k be the number of variables in V, and let v be a list of distinct variables, $\tilde{v}$ = V. Let G be a new k-place predicate symbol. By Theorem 5.4, we have to prove that

$$WP(S_1;S_2,G(v)) \Rightarrow WP(\underline{\text{rep}}\ \beta : S_1';S_2' \ \underline{\text{per}},\ G(v)).$$

First, we have that

$$WP(S_2,G(v)) \Longleftrightarrow \exists x(y=t \wedge I) \wedge \forall x(y=t \wedge I \Rightarrow WP(S_2',I \wedge G(v)[t/y])).$$

Denote the first conjunct of the right hand side P_1 and the second P_2. Then

$$WP(S_1,P_1 \wedge P_2) \iff \exists x(y=t \wedge I)$$

$$\wedge\ \forall x(y=t \wedge I \Rightarrow WP(S_1',I \wedge P_1[t/y] \wedge P_2[t/y])).$$

We concentrate on the formula $P_2[t/y]$. Changing the bound variable x to a fresh variable x' gives

$$P_2 \iff \forall x'(y=t[x'/x] \wedge I[x'/x] \Rightarrow WP(S_2',I \wedge G(v)[t/y])[x'/x]),$$

thus making t free for y in the formula P_2. Thus we have that

$$P_2[t/y] \iff \forall x'(t=t[x'/x] \wedge I[x'/x] \Rightarrow WP(S_2',I \wedge G(v)[t/y])[x'/x])$$

By substituting x for x' in $P_2[t/y]$, we get that

$$P_2[t/y] \Rightarrow (t=t \wedge I \Rightarrow WP(S_2',I \wedge G(v)[t/y]))\ ,$$

or equivalently,

$$P_2[t/y] \wedge I \Rightarrow WP(S_2',I \wedge G(v)[t/y]).$$

Using this result, we have that

$$I \wedge P_1[t/y] \wedge P_2[t/y] \Rightarrow WP(S_2',I \wedge G(v)[t/y]),$$

and using the generalization rule, this gives us that

$$\forall w(I \wedge P_1[t/y] \wedge P_2[t/y] \Rightarrow WP(S_2',I \wedge G(v)[t/y])),$$

where w is a list of distinct variables, $\tilde{w}$ = W. We may therefore use Lemma 5.11(i), and get

$$WP(S_1',I \wedge P_1[t/y] \wedge P_2[t/y]) \Rightarrow WP(S_1',WP(S_2',I \wedge G(v)[t/y]))$$

$$\iff WP(S_1';S_2',I \wedge G(v)[t/y]).$$

Thus we have that

$$WP(S_1;S_2',G(v)) \iff WP(S_1,P_1 \wedge P_2)$$

$$\iff \exists x(y=t \wedge I) \wedge \forall x(y=t \wedge I \Rightarrow WP(S_1',I \wedge P_1[t/y] \wedge P_2[t/y]))$$

$$\Rightarrow \exists x(y=t \wedge I) \wedge \forall x(y=t \wedge I \Rightarrow WP(S_1';S_2',I \wedge G(v)[t/y]))$$

$$\iff WP(\underline{rep}\ \beta : S_1';S_2',G(v)),$$

as required. □

LEMMA 7.8. *Assume that* S_i *and* S_i' *are as in* (7.9), $i = 1,\ldots,n$. *Denote*

$$S = \underline{if}\ B_1 \to S_1 \,\square \ldots \square\, B_n \to S_n\ \underline{fi}$$

$$S' = \underline{if}\ B_1' \to S_1' \,\square \ldots \square\, B_n' \to S_n'\ \underline{fi}.$$

Assume further that

$$\Delta \vdash P \Rightarrow \exists x(y=t \wedge I)$$

and

$$\Delta \vdash P \wedge y=t \wedge I \Rightarrow (B_i \iff B_i'), \quad \textit{for } i = 1,\ldots,n.$$

Then

$$\Delta \vdash \{P\};S \leq \underline{rep}\ \beta : S'\ \underline{per}.$$

PROOF. Let v and G be as in the Proof of 7.7, and assume that

$$WP(\{P\};S,G(v)),$$

i.e. writing BB for $B_1 \vee \ldots \vee B_n$, we have the assumption

$$(7.11) \qquad P \wedge BB \wedge \bigwedge_{1\leq i\leq n} (B_i \Rightarrow WP(S_i,G(v))).$$

We have to prove that $WP(\underline{rep}\ \beta : S'\ \underline{per}, G(v))$ holds, i.e. that

$$\exists x(y=t \wedge I) \wedge \forall x(y=t \wedge I \Rightarrow WP(S',I \wedge G(v)[t/y])).$$

The first conjunct is implied by (7.11) because of the assumption, so we only need to prove that the second conjunct also is implied. Assume therefore that

(7.12) $\quad y=t \wedge I.$

Then $B_i \Rightarrow B_i'$ by the assumption, for $i = 1,\ldots,n$, so we get from (7.11) that

$$B_1' \vee \ldots \vee B_n'.$$

Now, let i be an integer, $1 \leq i \leq n$, and assume that B_i'. By the assumption of the lemma, this means that B_i holds. By (7.11), this will again give that $WP(S_i,G(v))$ holds, i.e. we have that

$$\exists x(y=t \wedge I) \wedge \forall x(y=t \wedge I \Rightarrow WP(S_i',I \wedge G(v)[t/y])).$$

Thus, by assumption (7.9), we have that

$$WP(S_i',I \wedge G(v)[t/y]).$$

Removing the assumption that B_i' holds, this means that

$$B_i' \Rightarrow WP(S_i',I \wedge G(v)[t/y]),$$

and as i was arbitrarily chosen, $1 \leq i \leq n$, we have that

$$\bigwedge_{1\leq i\leq n} (B_i' \Rightarrow WP(S_i',I \wedge G(v)[t/y]).$$

This, together with the fact that $B_1' \vee \ldots \vee B_n'$ holds, means that

$$WP(S',I \wedge G(v)[t/y]).$$

Eliminating assumption (7.12) gives

$$y=t \wedge I \Rightarrow WP(S',I \wedge G(v)[t/y]),$$

and as x is not free in assumption (7.11), we have

$$\forall x(y=t \wedge I \Rightarrow WP(S',I \wedge G(v)[t/y])),$$

thus concluding the proof. □

LEMMA 7.9. *Assume that* S_i, S_i' *are as in* (7.9), i = 1,...,n. *Denote*

$$DO = \underline{do}\ B_1 \to S_1;\{P\} \square \ldots \square B_n \to S_n;\{P\}\ \underline{od},$$

$$DO' = \underline{do}\ B_1' \to S_1';\{P'\} \square \ldots \square B_n' \to S_n';\{P'\}\ \underline{od}$$

$$P' = P[t/y] \wedge I.$$

Assume further that

$$\Delta \vdash P \Rightarrow \exists x(y=t \wedge I)$$

and

$$\Delta \vdash P \wedge y=t \wedge I \Rightarrow (B_i \Leftrightarrow B_i'), \quad \textit{for } i = 1,\ldots,n.$$

Then

$$\Delta \vdash \{P\};DO \leq \underline{rep}\ \beta : DO'\ \underline{per}.$$

PROOF. Let DO^n and DO'^n have their usual meaning. We will prove that

$$\{P\};DO^n \leq \underline{rep}\ \beta : DO'^n\ \underline{per}, \quad \text{for } n < \omega. \tag{7.13}$$

Because $DO'^n \leq DO'$, this will give us that

$$\{P\};DO^n \leq \underline{rep}\ \beta : DO'\ \underline{per}, \quad \text{for } n < \omega,$$

from which the desired result then follows using Lemma 7.5.

For n = 0, (7.13) obviously holds, as DO^0 = abort. Assume that (7.13) holds for n, n ≥ 0. We have that

$$\begin{aligned}\{P\};DO^{n+1} &= \{P\};\ \underline{if}\ BB \to \underbrace{\underline{if}\ B_1 \to S_1;\{P\} \square \ldots \square B_n \to S_n;\{P\}\ \underline{fi}}_{DO^n};\\ &\qquad \square\ \sim BB \to skip\ \underline{fi}\\ &\leq \{P\};\ \underline{if}\ BB \to \{P\};\ \underline{if}\ B_1 \to S_1;\{P\} \square \ldots\\ &\qquad\qquad \square\ B_n \to S_n;\{P\}\ \underline{fi};\ \{P\};DO^n\\ &\qquad \square\ \sim BB \to \{P\};\ skip\ \underline{fi},\end{aligned}$$

using the rules for assertions of Section 7.2.

By Lemma 7.6, we have

(7.14) $\{P\}$; skip $\leq$ rep β : skip per.

Actually we have the stronger result that

$$\{P\}; \text{skip} \leq \underline{\text{rep}}\ \beta : \{P'\}; \text{skip}\ \underline{\text{per}}.$$

This means that

$$\{P\} \leq \underline{\text{rep}}\ \beta : \{P'\}\ \underline{\text{per}},$$

because $\{P\}$; skip $\approx \{P\}$ for any P.

Now, using Lemma 7.7 we get that

$$S_i;\{P\} \leq \underline{\text{rep}}\ \beta : S_i';\{P'\}\ \underline{\text{per}}, \quad \text{for } i = 1,\ldots,n.$$

And using Lemma 7.8, we get from this that

$$\{P\};\ \underline{\text{if}}\ B_1 \to S_1;\{P\} \,\square \ldots \square\, B_n \to S_n;\{P\}\ \underline{\text{fi}}$$

$$\leq \underline{\text{rep}}\ \beta : \underline{\text{if}}\ B_1' \to S_1';\{P'\} \,\square \ldots \square\, B_n' \to S_n';\{P'\}\ \underline{\text{fi}}\ \underline{\text{per}}$$

Finally, using induction hypothesis (7.13), result (7.14), Lemma 7.7 and Lemma 7.8 again, we get the result

$$\{P\};DO^{n+1} \leq \underline{\text{rep}}\ \beta : DO'^{n+1}\ \underline{\text{per}},$$

thus proving that (7.13) holds for each $n < \omega$. □

LEMMA 7.10. *Let* V *be* V' $\cup\ \tilde{y}'$, V' $\cap\ \tilde{y}' = \emptyset$, *for some list* y' *of program variables and some nonempty set* V' *of program variables. Assume that* $\tilde{y} \subseteq \tilde{y}'$. *Then*

$$\Delta \vdash \underline{\text{beg}}\ y' : \underline{\text{beg}}\ \beta : S\ \underline{\text{per}}\ \underline{\text{end}} \leq \underline{\text{beg}}\ y'',x : S\ \underline{\text{end}},$$

where $\tilde{y}'' = \tilde{y}' - \tilde{y}$, S: W $\to$ W *and* β = x/y.(y=t $\wedge$ I): V $\to$ W *as before.*

PROOF. Let k be the number of variables in V', and let v' be a list of distinct variables, $\tilde{v}' = V'$. Let G be a k-place predicate symbol. Let S_1 denote the left hand side of the above and S_2 the right hand side. We have to prove that

$$WP(S_1,G(v')) \Rightarrow WP(S_2,G(v')).$$

Assume therefore that

$$WP(S_1,G(v')).$$

The assumption gives us, by definition of WP, that

$$\forall y'WP(\underline{beg}\ \beta : S\ \underline{per},\ G(v')),$$

which again is equivalent to

$$\forall y'xWP(S,I \wedge G(v')[t/y]).$$

Now, because $\tilde{y} \cap V' = \emptyset$, as $\tilde{y} \subseteq \tilde{y}'$, y cannot occur free in G(v'), so $G(v')[t/y] = G(v')$. Thus we have that

$$\forall w(I \wedge G(v')[t/y] \Rightarrow G(v')),$$

and using Lemma 5.11(i), this gives us that

$$WP(S,I \wedge G(v')[t/y]) \Rightarrow WP(S,G(v')),$$

i.e. the assumption gives us that

$$\forall y'xWP(S,G(v')).$$

As y cannot occur free in WP(S,G(v')), because $S: W \to W$, and $\tilde{y} \cap W = \emptyset$, this is again equivalent to

$$\forall y''xWP(S,G(v')),$$

which, from the definition of WP, is

$$WP(\underline{beg}\ y'',x:\ S\ \underline{end},\ G(v')).$$

This proves the lemma. □

The example program can now be handled by the following rules of inference. The soundness of these rules are immediate consequences of the lemmas proved above (β and $S_1',\ldots,S_n'$ are assumed to be as in (7.9), while $S_1,\ldots,S_n$ are any program descriptions in V).

1. Composition

(i)
$$\frac{S_1 \le \underline{rep}\ \beta:\ S_1'\ \underline{per},\ S_2 \le \underline{rep}\ \beta:\ S_2'\ \underline{per}}{S_1;S_2 \le \underline{rep}\ \beta:\ S_1';S_2'\ \underline{per}}$$

(ii)
$$\frac{S_1 \le \underline{beg}\ \beta:\ S_1'\ \underline{per},\ S_2 \le \underline{rep}\ \beta:\ S_2'\ \underline{per}}{S_1;S_2 \le \underline{beg}\ \beta:\ S_1';S_2'\ \underline{per}}$$

2. Selection

$$\frac{S_i \le \underline{rep}\ \beta:\ S_i'\ \underline{per},\quad i=1,\ldots,n,\ H_1,\ H_2}{\begin{array}{l}\{P\};\ \underline{if}\ B_1 \to S_1 \,\square\, \ldots \,\square\, B_n \to S_n\ \underline{fi} \\ \quad \le \underline{rep}\ \beta: \\ \qquad \underline{if}\ B_1' \to S_1' \,\square\, \ldots \,\square\, B_n' \to S_n'\ \underline{fi} \\ \quad \underline{per},\end{array}}$$

where

$$H_1:\ P \Rightarrow \exists x(y=t \wedge I)$$

and

$$H_2:\ (P \wedge y=t \wedge I) \Rightarrow (B_i \Leftrightarrow B_i'),\quad i=1,\ldots,n.$$

3. Iteration

$$\frac{S_i \le \underline{rep}\ \beta\colon S_i'\ \underline{per},\quad i = 1,\ldots,n,\ H_1,\ H_2}{\begin{array}{l}\{P\};\ \underline{do}\ B_1 \to S_1;\{P\}\square\ldots\square B_n \to S_n;\{P\}\ \underline{od}\\ \quad \le \underline{rep}\ \beta\colon\\ \qquad \underline{do}\ B_1' \to S_1' \square \ldots \square B_n' \to S_n'\ \underline{od}\\ \quad \underline{per}\end{array}}$$

where H_1 and H_2 are as above

4. Blocks

$$\frac{S \le \underline{beg}\ \beta\colon S'\ \underline{per}}{\begin{array}{l}\underline{beg}\ y,z\colon S\ \underline{end}\\ \le \underline{beg}\ x,z\colon S'\ \underline{end}.\end{array}}$$

The transition steps by which the data representation of version B_2 in Section 7.1 is changed can be done with these proof rules. In order to apply them we need to prove the conditions

$$H_1\colon\ R_2 \Rightarrow \exists x,y.(h=x^y \wedge x > 1),$$

and

$$H_2\colon\ R_2 \wedge h = x^y \wedge x > 1 \Rightarrow (h\neq 1 \iff y\neq 0).$$

Writing R_2 explicitly, we get the formulas

$$h\cdot z = X^Y \wedge h \ge 1 \Rightarrow \exists x,y.(h=x^y \wedge x > 1)$$

and

$$h\cdot z = X^Y \wedge h \ge 1 \wedge h=x^y \wedge x > 1 \Rightarrow (h\neq 1 \iff y\neq 0).$$

These are readily seen to be true.

REFERENCES

[1] BACK, R.J.R., *On the correctness refinement steps in program development,* Report A-1978-4, Dept. of Computer Science, Univ. of Helsinki, 1978.

[2] BACK, R.J.R., *On the notion of correct refinement of programs,* in Proc. 5th Scandinavian Logic Symposium (F.V. Jansen, B.H. Mayoh and K.K. Møller, eds.), Aalborg University Press, 1979, (to appear in Journal of Computer and System Sciences).

[3] BACK, R.J.R., *Proving total correctness of nondeterministic programs in infinitary logic,* to appear in Acta Informatica.

[4] BACK, R.J.R., *Semantics of unbounded nondeterminism,* in Proc. 7th Coll. Automata, Languages and Programming, (J.W. de Bakker and J. van Leeuwen, eds), Lecture Notes in Computer Science 85, Springer, 1980.

[5] BANACHOWSKI, L., A. KRECZMAR, G. MIRKOWSKA, H. RASIOWA & A. SALWICKI, *An introduction to algorithmic logic; metamathematical investigations in the theory of programs,* in Mathematical Foundations of Computer Science, Banach Center Publications, (A. Mazurkiewicz and Z. Pawlak, eds.), pp. 7-99, Warsaw, 1977.

[6] BAUER, F.L., *Program development by stepwise transformations - The project CIP,* in Program Construction, (F.L. Bauer and M. Broy, eds.), pp. 237-272, Lecture Notes in Computer Science 69, Springer, 1979.

[7] BAUER, F.L., M. BROY, H. PARTSCH, P. PEPPER & H. WOSSNER, *Systematics of transformation rules,* in Program Construction, (F.L. Bauer and M. Broy, eds.), pp. 273-289, Lecture Notes in Computer Science 69, Springer, 1979.

[8] BOOM, H.J., *A weaker precondition for loops,* Report IW 104/78, Mathematisch Centrum, 1978.

[9] BROY, M., R. GNATZ & M. WIRSING, *Semantics of nondeterministic and noncontinuous constructs,* in Program Construction, (F.L. Bauer and M. Broy, eds.), pp. 553-592, Lecture Notes in Computer Science 69, Springer, 1979.

[10] BURSTALL, R.M. & J. DARLINGTON, *Some transformations for developing recursive programs,* Journal of ACM 24, 1, pp. 44-67, 1977.

[11] CORRELL, C.H., *Proving programs correct through refinement,* Acta Informatica 9, pp. 121-132, 1978.

[12] DE BAKKER, J.W., Mathematical Theory of Program Correctness, Prentice-Hall, 1980.

[13] DIJKSTRA, E.W., *A constructive approach to the problem of program correctness,* BIT 8, pp. 174-186, 1968.

[14] DIJKSTRA, E.W., *Notes on structured programming,* in Dahl, O.J., E.W. Dijkstra and C.A.R. Hoare: Structured Programming, Academic Press, 1971.

[15] DIJKSTRA, E.W., *A Discipline of Programming,* Prentice-Hall, 1976.

[16] DIJKSTRA, E.W., Private communication, 1978.

[17] ENGELER, E., *Remarks on the theory of geometrical constructions,* in: *The Syntax and Semantics of Infinitary Languages,* (J. Barwise, ed.), Lecture Notes in Mathematics 72, Springer, 1968.

[18] ENGELER, E., *Algorithmic Logic,* in: *Foundations of Computer Science,* (J.W. de Bakker, ed.), pp. 57-85, Mathematical Centre Tracts 63, Mathematisch Centrum, 1975.

[19] FEFERMAN, S., *Lectures in Proof Theory,* in Proc. Summer School in Mathematical Logic, (M.H. Löb, ed.), pp. 1-108, Springer, 1968.

[20] GERHART, S.L., *Correctness preserving program transformations,* in Proc. Second ACM Conference on Principles of Programming Languages, pp. 54-66, 1975.

[21] HAREL, D., *First-Order Dynamic Logic,* Lecture Notes in Computer Science 68, Springer, 1979.

[22] HAREL, D., A. PNUELI & J. STAVI, *A complete axiomatic system for proving deductions about recursive programs,* in Proc. 9th Annual ACM Symp. on the Theory of Computing, 1977.

[23] HOARE, C.A.R., *An axiomatic basis for computer programming,* Communications ACM 12, 10, pp. 576-580, 1969.

[24] HOARE, C.A.R., *Procedures and parameters: An axiomatic approach,* in Symposium on Semantics and Algorithmic Languages, (E. Engeler, ed.), pp. 102-116, Lecture Notes in Mathematics 188, Springer, 1971.

[25] HOARE, C.A.R., *Proof of correctness of data representation,* Acta Informatica 1, 4, pp. 271-281, 1972.

[26] HOARE, C.A.R., *Some properties of predicate transformers,* Journal of ACM, 25, 3, July 1978.

[27] KARP, C.R., *Languages with Expressions of Infinite Length,* North-Holland, 1964.

[28] KATZ, S.M. & Z. MANNA, *Logical analysis of programs,* Communications of ACM 19, 4, pp. 188-206, 1976.

[29] KEISLER, H.J., *Model Theory for Infinitary Logic,* North-Holland, 1971.

[30] KNUTH, D.E., *Structured programming with the goto statement,* Computing Surveys 6, 4, pp. 261-301, 1974.

[31] LAMPSON, B.W., J.J. HORNING, R.L. LONDON, J.G. MITCHELL & J. POPEK, *Report on the programming language Euclid,* Sigplan Notices 12, 2, 1977.

[32] LISKOV, B.H., A. SNYDER, R. ATKINSON & C. SCHAFFERT, *Abstraction mechanism in CLU,* Communications of ACM 20, 8, pp. 564-576, 1977.

[33] LOVEMAN, D.B., *Program improvement by source-to-source transformations,* Journal of ACM 24, 1, pp. 121-145, 1977.

[34] MANNA, Z., *Mathematical Theory of Computing,* McGraw-Hill, 1974.

[35] MEERTENS, L.G.L.T., *Abstracto 84: The next generation,* Report IW 120/79, Mathematisch Centrum, 1979.

[36] MILNER, R., *An algebraic definition of simulation between programs,* Report CS 205, Dept. of Comp. Science, Stanford Univ., 1971.

[37] PLOTKIN, G.D., *A power-domain construction,* SIAM Journal of Computing 5, 3, pp. 452-487, 1976.

[38] PRATT, V.R., *Semantic considerations of Floyd-Hoare logic,* in Proc. 17th IEEE Symp. on Foundations of Computer Science, pp. 109-121, 1976.

[39] SALWICKI, A., *Formalized algorithmic languages,* Bull. Acad. Polon. Sci., Ser. Math. 18, pp. 227-232, 1970.

[40] SCOTT, D., *Logic with denumerably long formulas and finite strings of quantifiers,* In Symp. on the Theory of Models, (J. Addison, L. Henkin and A. Tarski, eds.), pp. 329-341, North-Holland, 1965.

[41] SMYTH, M.B., *Power domains,* Journal of Computer and System Sciences 16, pp. 23-36, 1978.

[42] WEGBREIT, B., *Goal directed program transformations,* IEEE Trans. on Software Engineering SE-2, 2, pp. 69-80, 1967.

[43] WIRTH, N., *Program development by stepwise refinement,* Communications of ACM 14, 4, pp. 221-227, 1971.

[44] WIRTH, N., *Systematic Programming,* Prentice-Hall, 1973.

[45] WIRTH, N., *Modula, a language for modular multiprogramming,* Software Practice and Experience 7, 1, 1977.

[46] WULFF, W.A., R.L. LONDON & M. SHAW, *An introduction to the construction and verification of Alphard programs,* IEEE Trans. on Software Engineering se-2, 4, pp. 253-265, 1976.

INDEX OF NOTATIONS

INDEX OF TERMS

TITLES IN THE SERIES MATHEMATICAL CENTRE TRACTS

(An asterisk before the MCT number indicates that the tract is under preparation).

A leaflet containing an order form and abstracts of all publications mentioned below is available at the Mathematisch Centrum, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. Orders should be sent to the same address.

MCT 1 T. VAN DER WALT, *Fixed and almost fixed points*, 1963. ISBN 90 6196 002 9.

MCT 2 A.R. BLOEMENA, *Sampling from a graph*, 1964. ISBN 90 6196 003 7.

MCT 3 G. DE LEVE, *Generalized Markovian decision processes, part I: Model and method*, 1964. ISBN 90 6196 004 5.

MCT 4 G. DE LEVE, *Generalized Markovian decision processes, part II: Probabilistic background*, 1964. ISBN 90 6196 005 3.

MCT 5 G. DE LEVE, H.C. TIJMS & P.J. WEEDA, *Generalized Markovian decision processes, Applications*, 1970. ISBN 90 6196 051 7.

MCT 6 M.A. MAURICE, *Compact ordered spaces*, 1964. ISBN 90 6196 006 1.

MCT 7 W.R. VAN ZWET, *Convex transformations of random variables*, 1964. ISBN 90 6196 007 X.

MCT 8 J.A. ZONNEVELD, *Automatic numerical integration*, 1964. ISBN 90 6196 008 8.

MCT 9 P.C. BAAYEN, *Universal morphisms*, 1964. ISBN 90 6196 009 6.

MCT 10 E.M. DE JAGER, *Applications of distrubutions in mathematical physics*, 1964. ISBN 90 6196 010 X.

MCT 11 A.B. PAALMAN-DE MIRANDA, *Topological semigroups*, 1964. ISBN 90 6196 011 8.

MCT 12 J.A.Th.M. VAN BERCKEL, H. BRANDT CORSTIUS, R.J. MOKKEN & A. VAN WIJNGAARDEN, *Formal properties of newspaper Dutch*, 1965. ISBN 90 6196 013 4.

MCT 13 H.A. LAUWERIER, *Asymptotic expansions*, 1966, out of print; replaced by MCT 54.

MCT 14 H.A. LAUWERIER, *Calculus of variations in mathematical physics*, 1966. ISBN 90 6196 020 7.

MCT 15 R. DOORNBOS, *Slippage tests*, 1966. ISBN 90 6196 021 5.

MCT 16 J.W. DE BAKKER, *Formal definition of programming languages with an application to the definition of ALGOL 60*, 1967. ISBN 90 6196 022 3.

MCT 17 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 1*, 1968. ISBN 90 6196 025 8.

MCT 18 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 2*, 1968. ISBN 90 6196 038 X.

MCT 19 J. VAN DER SLOT, *Some properties related to compactness*, 1968. ISBN 90 6196 026 6.

MCT 20 P.J. VAN DER HOUWEN, *Finite difference methods for solving partial differential equations*, 1968. ISBN 90 6196 027 4.

MCT 21 E. WATTEL, *The compactness operator in set theory and topology*, 1968. ISBN 90 6196 028 2.

MCT 22 T.J. DEKKER, *ALGOL 60 procedures in numerical algebra, part 1*, 1968. ISBN 90 6196 029 0.

MCT 23 T.J. DEKKER & W. HOFFMANN, *ALGOL 60 procedures in numerical algebra, part 2*, 1968. ISBN 90 6196 030 4.

MCT 24 J.W. DE BAKKER, *Recursive procedures*, 1971. ISBN 90 6196 060 6.

MCT 25 E.R. PAËRL, *Representations of the Lorentz group and projective geometry*, 1969. ISBN 90 6196 039 8.

MCT 26 EUROPEAN MEETING 1968, *Selected statistical papers, part I*, 1968. ISBN 90 6196 031 2.

MCT 27 EUROPEAN MEETING 1968, *Selected statistical papers, part II*, 1969. ISBN 90 6196 040 1.

MCT 28 J. OOSTERHOFF, *Combination of one-sided statistical tests*, 1969. ISBN 90 6196 041 X.

MCT 29 J. VERHOEFF, *Error detecting decimal codes*, 1969. ISBN 90 6196 042 8.

MCT 30 H. BRANDT CORSTIUS, *Exercises in computational linguistics*, 1970. ISBN 90 6196 052 5.

MCT 31 W. MOLENAAR, *Approximations to the Poisson, binomial and hypergeometric distribution functions*, 1970. ISBN 90 6196 053 3.

MCT 32 L. DE HAAN, *On regular variation and its application to the weak convergence of sample extremes*, 1970. ISBN 90 6196 054 1.

MCT 33 F.W. STEUTEL, *Preservation of infinite divisibility under mixing and related topics*, 1970. ISBN 90 6196 061 4.

MCT 34 I. JUHÁSZ, A. VERBEEK & N.S. KROONENBERG, *Cardinal functions in topology*, 1971. ISBN 90 6196 062 2.

MCT 35 M.H. VAN EMDEN, *An analysis of complexity*, 1971. ISBN 90 6196 063 0.

MCT 36 J. GRASMAN, *On the birth of boundary layers*, 1971. ISBN 90 6196 064 9.

MCT 37 J.W. DE BAKKER, G.A. BLAAUW, A.J.W. DUIJVESTIJN, E.W. DIJKSTRA, P.J. VAN DER HOUWEN, G.A.M. KAMSTEEG-KEMPER, F.E.J. KRUSEMAN ARETZ, W.L. VAN DER POEL, J.P. SCHAAP-KRUSEMAN, M.V. WILKES & G. ZOUTENDIJK, *MC-25 Informatica Symposium* 1971. ISBN 90 6196 065 7.

MCT 38 W.A. VERLOREN VAN THEMAAT, *Automatic analysis of Dutch compound words*, 1971. ISBN 90 6196 073 8.

MCT 39 H. BAVINCK, *Jacobi series and approximation*, 1972. ISBN 90 6196 074 6.

MCT 40 H.C. TIJMS, *Analysis of (s,S) inventory models*, 1972. ISBN 90 6196 075 4.

MCT 41 A. VERBEEK, *Superextensions of topological spaces*, 1972. ISBN 90 6196 076 2.

MCT 42 W. VERVAAT, *Success epochs in Bernoulli trials (with applications in number theory)*, 1972. ISBN 90 6196 077 0.

MCT 43 F.H. RUYMGAART, *Asymptotic theory of rank tests for independence*, 1973. ISBN 90 6196 081 9.

MCT 44 H. BART, *Meromorphic operator valued functions*, 1973. ISBN 90 6196 082 7.

MCT 45 A.A. BALKEMA, *Monotone transformations and limit laws* 1973. ISBN 90 6196 083 5.

MCT 46 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 1: The language*, 1973. ISBN 90 6196 084 3.

MCT 47 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 2: The compiler*, 1973. ISBN 90 6196 085 1.

MCT 48 F.E.J. KRUSEMAN ARETZ, P.J.W. TEN HAGEN & H.L. OUDSHOORN, *An ALGOL 60 compiler in ALGOL 60, Text of the MC-compiler for the EL-X8*, 1973. ISBN 90 6196 086 X.

MCT 49 H. KOK, *Connected orderable spaces*, 1974. ISBN 90 6196 088 6.

MCT 50 A. VAN WIJNGAARDEN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER (Eds), *Revised report on the algorithmic language ALGOL 68*, 1976. ISBN 90 6196 089 4.

MCT 51 A. HORDIJK, *Dynamic programming and Markov potential theory*, 1974. ISBN 90 6196 095 9.

MCT 52 P.C. BAAYEN (ed.), *Topological structures*, 1974. ISBN 90 6196 096 7.

MCT 53 M.J. FABER, *Metrizability in generalized ordered spaces*, 1974. ISBN 90 6196 097 5.

MCT 54 H.A. LAUWERIER, *Asymptotic analysis, part 1*, 1974. ISBN 90 6196 098 3.

MCT 55 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 1: Theory of designs, finite geometry and coding theory*, 1974. ISBN 90 6196 099 1.

MCT 56 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 2: Graph theory, foundations, partitions and combinatorial geometry*, 1974. ISBN 90 6196 100 9.

MCT 57 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 3: Combinatorial group theory*, 1974. ISBN 90 6196 101 7.

MCT 58 W. ALBERS, *Asymptotic expansions and the deficiency concept in statistics*, 1975. ISBN 90 6196 102 5.

MCT 59 J.L. MIJNHEER, *Sample path properties of stable processes*, 1975. ISBN 90 6196 107 6.

MCT 60 F. GÖBEL, *Queueing models involving buffers*, 1975. ISBN 90 6196 108 4.

*MCT 61 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 1*, ISBN 90 6196 109 2.

*MCT 62 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 2*, ISBN 90 6196 110 6.

MCT 63 J.W. DE BAKKER (ed.), *Foundations of computer science*, 1975. ISBN 90 6196 111 4.

MCT 64 W.J. DE SCHIPPER, *Symmetric closed categories*, 1975. ISBN 90 6196 112 2.

MCT 65 J. DE VRIES, *Topological transformation groups 1 A categorical approach*, 1975. ISBN 90 6196 113 0.

MCT 66 H.G.J. PIJLS, *Locally convex algebras in spectral theory and eigenfunction expansions*, 1976. ISBN 90 6196 114 9.

*MCT 67 H.A. LAUWERIER, *Asymptotic analysis, part 2*, ISBN 90 6196 119 X.

MCT 68 P.P.N. DE GROEN, *Singularly perturbed differential operators of second order*, 1976. ISBN 90 6196 120 3.

MCT 69 J.K. LENSTRA, *Sequencing by enumerative methods*, 1977. ISBN 90 6196 125 4.

MCT 70 W.P. DE ROEVER JR., *Recursive program schemes: Semantics and proof theory*, 1976. ISBN 90 6196 127 0.

MCT 71 J.A.E.E. VAN NUNEN, *Contracting Markov decision processes*, 1976. ISBN 90 6196 129 7.

MCT 72 J.K.M. JANSEN, *Simple periodic and nonperiodic Lamé functions and their applications in the theory of conical waveguides*, 1977. ISBN 90 6196 130 0.

MCT 73 D.M.R. LEIVANT, *Absoluteness of intuitionistic logic*, 1979. ISBN 90 6196 122 X.

MCT 74 H.J.J. TE RIELE, *A theoretical and computational study of generalized aliquot sequences*, 1976. ISBN 90 6196 131 9.

MCT 75 A.E. BROUWER, *Treelike spaces and related connected topological spaces*, 1977. ISBN 90 6196 132 7.

MCT 76 M. REM, *Associations and the closure statement*, 1976. ISBN 90 6196 135 1.

MCT 77 W.C.M. KALLENBERG, *Asymptotic optimality of likelihood ratio tests in exponential families*, 1977. ISBN 90 6196 134 3.

MCT 78 E. DE JONGE & A.C.M. VAN ROOIJ, *Introduction to Riesz spaces*, 1977. ISBN 90 6196 133 5.

MCT 79 M.C.A. VAN ZUIJLEN, *Empirical distributions and rank statistics*, 1977. ISBN 90 6196 145 9.

MCT 80 P.W. HEMKER, *A numerical study of stiff two-point boundary problems*, 1977. ISBN 90 6196 146 7.

MCT 81 K.R. APT & J.W. DE BAKKER (Eds), *Foundations of computer science II*, part 1, 1976. ISBN 90 6196 140 8.

MCT 82 K.R. APT & J.W. DE BAKKER (Eds), *Foundations of computer science II*, part 2, 1976. ISBN 90 6196 141 6.

MCT 83 L.S. BENTHEM JUTTING, *Checking Landau's "Grundlagen" in the AUTOMATH system*, 1979. ISBN 90 6196 147 5.

MCT 84 H.L.L. BUSARD, *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?) books vii-xii*, 1977. ISBN 90 6196 148 3.

MCT 85 J. VAN MILL, *Supercompactness and Wallman spaces*, 1977. ISBN 90 6196 151 3.

MCT 86 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix I, A programming system for operations on vectors and matrices over arbitrary fields and of variable size*. 1978. ISBN 90 6196 152 1.

*MCT 87 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix II*, ISBN 90 6196 153 X.

MCT 88 A. SCHRIJVER, *Matroids and linking systems*, 1977. ISBN 90 6196 154 8.

MCT 89 J.W. DE ROEVER, *Complex Fourier transformation and analytic functionals with unbounded carriers*, 1978. ISBN 90 6196 155 6.

*MCT 90 L.P.J. GROENEWEGEN, *Characterization of optimal strategies in dynamic games*, . ISBN 90 6196 156 4.

MCT 91 J.M. GEYSEL, *Transcendence in fields of positive characteristic*, 1979. ISBN 90 6196 157 2.

MCT 92 P.J. WEEDA, *Finite generalized Markov programming*, 1979. ISBN 90 6196 158 0.

MCT 93 H.C. TIJMS & J. WESSELS (eds), *Markov decision theory*, 1977. ISBN 90 6196 160 2.

MCT 94 A. BIJLSMA, *Simultaneous approximations in transcendental number theory*, 1978. ISBN 90 6196 162 9.

MCT 95 K.M. VAN HEE, *Bayesian control of Markov chains*, 1978. ISBN 90 6196 163 7.

MCT 96 P.M.B. VITÁNYI, *Lindenmayer systems: Structure, languages, and growth functions*, 1980. ISBN 90 6196 164 5.

*MCT 97 A. FEDERGRUEN, *Markovian control problems; functional equations and algorithms*, . ISBN 90 6196 165 3.

MCT 98 R. GEEL, *Singular perturbations of hyperbolic type*, 1978. ISBN 90 6196 166 1.

MCT 99 J.K. LENSTRA, A.H.G. RINNOOY KAN & P. VAN EMDE BOAS, *Interfaces between computer science and operations research*, 1978. ISBN 90 6196 170 X.

MCT 100 P.C. BAAYEN, D. VAN DULST & J. OOSTERHOFF (Eds), *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1*, 1979. ISBN 90 6196 168 8.

MCT 101 P.C. BAAYEN, D. VAN DULST & J. OOSTERHOFF (Eds), *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2*, 1979. ISBN 90 6196 169 6.

MCT 102 D. VAN DULST, *Reflexive and superreflexive Banach spaces*, 1978. ISBN 90 6196 171 8.

MCT 103 K. VAN HARN, *Classifying infinitely divisible distributions by functional equations*, 1978. ISBN 90 6196 172 6.

MCT 104 J.M. VAN WOUWE, *Go-spaces and generalizations of metrizability*, 1979. ISBN 90 6196 173 4.

*MCT 105 R. HELMERS, *Edgeworth expansions for linear combinations of order statistics*, . ISBN 90 6196 174 2.

MCT 106 A. SCHRIJVER (Ed.), *Packing and covering in combinatorics*, 1979. ISBN 90 6196 180 7.

MCT 107 C. DEN HEIJER, *The numerical solution of nonlinear operator equations by imbedding methods*, 1979. ISBN 90 6196 175 0.

MCT 108 J.W. DE BAKKER & J. VAN LEEUWEN (Eds), *Foundations of computer science III*, part 1, 1979. ISBN 90 6196 176 9.

MCT 109 J.W. DE BAKKER & J. VAN LEEUWEN (Eds), *Foundations of computer science III*, part 2, 1979. ISBN 90 6196 177 7.

MCT 110 J.C. VAN VLIET, *ALGOL 68 transput*, part I: *Historical review and discussion of the implementation model*, 1979. ISBN 90 6196 178 5.

MCT 111 J.C. VAN VLIET, *ALGOL 68 transput*, part II: *An implementation model*, 1979. ISBN 90 6196 179 3.

MCT 112 H.C.P. BERBEE, *Random walks with stationary increments and renewal theory*, 1979. ISBN 90 6196 182 3.

MCT 113 T.A.B. SNIJDERS, *Asymptotic optimality theory for testing problems with restricted alternatives*, 1979. ISBN 90 6196 183 1.

MCT 114 A.J.E.M. JANSSEN, *Application of the Wigner distribution to harmonic analysis of generalized stochastic processes*, 1979. ISBN 90 6196 184 X.

MCT 115 P.C. BAAYEN & J. VAN MILL (Eds), *Topological Structures II*, part 1, 1979. ISBN 90 6196 185 5.

MCT 116 P.C. BAAYEN & J. VAN MILL (Eds), *Topological Structures II*, part 2, 1979. ISBN 90 6196 186 6.

MCT 117 P.J.M. KALLENBERG, *Branching processes with continuous state space*, 1979. ISBN 90 6196 188 2.

MCT 118 P. GROENEBOOM, *Large deviations and asymptotic efficiencies*, 1980. ISBN 90 6196 190 4.

MCT 119 F. J. PETERS, *Sparse matrices and substructures, with a novel implementation of finite element algorithms*, 1980. ISBN 90 6196 192 0.

MCT 120 W.P.M. DE RUYTER, *On the asymptotic analysis of large-scale ocean circulation*, 1980. ISBN 90 6196 192 9.

MCT 121 W.H. HAEMERS, *Eigenvalue techniques in design and graph theory*, 1980. ISBN 90 6196 194 7.

MCT 122 J.C.P. BUS, *Numerical solution of systems of nonlinear equations*, 1980. ISBN 90 6196 195 5.

MCT 123 I. YUHÁSZ, *Cardinal functions in topology - ten years later*, 1980. ISBN 90 6196 196 3.

MCT 124 R.D. GILL, *Censoring and stochastic integrals*, 1980. ISBN 90 6196 197 1.

MCT 125 R. EISING, *2-D systems, an algebraic approach*, 1980. ISBN 90 6196 198 X.

MCT 126 G. VAN DER HOEK, *Reduction methods in nonlinear programming*, 1980. ISBN 90 6196 199 8.

MCT 127 J.W. KLOP, *Combinatory reduction systems*, 1980. ISBN 90 6196 200 5.

MCT 128 A.J.J. TALMAN, *Variable dimension fixed point algorithms and triangulations*, 1980. ISBN 90 6196 201 3.

MCT 129 G. VAN DER LAAN, *Simplicial fixed point algorithms*, 1980. ISBN 90 6196 202 1.

MCT 130 P.J.W. TAN HAGEN et al., *ILP Intermediate language for pictures*, 1980. ISBN 90 6196 204 8.

MCT 131 R.J.R. BACK, *Correctness preserving program refinements: Proof theory and applications*, 1980. ISBN 90 6196 207 2.

MCT 132 H.M. MULDER, *The interval function of a graph*, 1980. ISBN 90 6196 208 0.

MCT 133 C.A.J. KLASSEN, *Statistical performance of location estimators*, 1981. ISBN 90 6196 209 9.