On the Correctness of Refinement Steps in Program Development

Ralph-Johan Back
Åbo Akademi University

ON THE CORRECTNESS OF REFINEMENT STEPS IN PROGRAM DEVELOPMENT

RALPH-JOHAN BACK

To be presented, with the permission of the Section of Mathematics and Natural Sciences of the Philosophical Faculty of the University of Helsinki, for public criticism in Auditorium III, Porthania, on December 2nd, 1978, at 10 o'clock a.m.

Helsinki 9.10.1978

Report A-1978-4

Department of Computer Science
University of Helsinki
Töölönkatu 11
SF-00100 Helsinki 10 Finland

ISBN 951-45-1505-6
ISSN 0356-6951

ABSTRACT

Stepwise refinement is here regarded as a technique for constructing correct programs. The main problem considered is how the correctness of an individual refinement step can be proved. For this purpose, a language of *descriptions* is defined in which both the programs and their specifications can be expressed. Correct refinement is then introduced as a binary relation of *refinement* between descriptions. Total correctness of programs will be a special case of correct refinement.

The first main result is a general proof rule by which refinement between descriptions can be established. The proof rule is formulated in the infinitary logic $L\omega_1\omega$, and its soundness and completeness proved.

The second main result consists in showing how stepwise refinement of programs can be carried out using descriptions. It will also be shown how stronger proof rules can be derived from the general proof rule for refinement in order to handle commonly occurring situations in program development such as operational and representational abstraction, applications of program transformation rules and introducing assertions into programs.

CR Categories: 5.24, 4.0, 5.21

AKNOWLEDGEMENT

My special thanks go to Professor Reino Kurki-Suonio for his encouragement and support troughout the work on this thesis, and to Ms Maaret Karttunen for the many discussions and argumentations, in which the ideas of this thesis were clarified.

I wish to thank Professor Martti Tienari for his help and his interest in my work and Doctor Esko Ukkonen for his very useful comments on the thesis and on an earlier version of it. I also want to thank Mr Kai Koskimies and Doctor Juha Oikkonen for their comments on the thesis. I am grateful for the opportunity to discuss some of these ideas with Professors Jaco de Bakker and Ole-Johan Dahl.

I wish to thank Mr Lars Backström for his support and for the generosity he has shown, by giving me plenty of time to concentrate on my own work. The same holds for all the people in the Computing Centre, who have provided a warm and friendly atmosphere to work in.

The financial support of the Magnus Ehrnroth Foundation and Svenska Vetenskapliga Centralrådet is also gratefully aknowledged.

Finally I want to thank my wife Barbro for doing far more than her share in a difficult time, and my little daughter Pia for all the peaceful nights that she allowed us to have.

CONTENTS

1. INTRODUCTION

The *stepwise refinement* method, developed primarily by Dijkstra[68,72,76] and Wirth[71,73], is nowadays an important and well-established program construction technique. The basic idea of this method 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 in Dijkstra[68] 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 structures or control structures of the program, as well as applications of general program transformation rules.

In this thesis we will consider the problem of establishing the correctness of a refinement step. A formal system is presented in which correctness of refinement 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 characterised by listing some of the more fundamental 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 refinements in a broad sense, including not only implementations of procedures but also refinements concerned with the data representations and control structures of the program, as well as the use of program transformations.

(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 reasoning on semantic considerations.

(5) We did not want to invent a formal system of our 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[69,71,72] agrees with point (2) above, except for program transformations (which are treated in his style in Gerhardt[75]), and also with (4), but only partially with (1) and (3) and not at all with (5) and (6). Harel & al[77] 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.

The language to be defined will contain a new kind of primitive statement called an *atomic description*. It can be loosely characterised as a non-deterministic assignment statement with an associated change of scope (i.e. a change of the set of variables available). The set of *descriptions* will be constructed out of the atomic descriptions using standard control structures such as composition, selection and iteration. We also have a nondeterministic binary choice statement. 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,- an assertion 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 transitive, thus 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, then the fact that each refinement is correctly done means that

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

for i = 0,1, ...,n-1. Transitivity then gives us that

$$S_0 \leq S_n,$$

i.e. that 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 shall also define an equivalence relation between descriptions obtained by requiring mutual refinement between descriptions. Again in chapter 4 we shall give a characterisation of refinement using weakest preconditions.

In chapter 5 a general proof rule for proving refinement between descriptions will be presented, and the soundness and completeness of this proof rule will be shown. Essentially we will prove $S \leq S'$ for descriptions S and S' by computing a corresponding formula of $L\omega_1\omega$, and then prove this formula using the axioms and inference rules of $L\omega_1\omega$. The weakest preconditions of descriptions will be needed in order to compute the formula corresponding to $S \leq S'$. The fact that the proof rule given is complete means that we may restrict ourselves to formal proofs in $L\omega_1\omega$ altogether, i.e. we may ignore semantical considerations. Some important properties of weakest preconditions and the refinement relation will also be proved in chapter 5.

In chapter 6 we go on to show how stepwise refinement is carried out using descriptions. We will show how to achieve 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. We will also define a subset of descriptions which will be called *program descriptions*, and provide some syntactic sugar for these. They are not as general as descriptions, but more convienient to work with in program development.

Finally, in chapter 7 we give an example of formal program development using program descriptions. We will give special proof rules for handling commonly occurring refinement steps, such as procedure implementations, introducing assertions into program descriptions, handling representational abstraction and changing the control structure of a program description. 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.

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 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 $L\alpha\beta$. $L\alpha\beta$ 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 [64], 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 in Karp [64]. The logic $L\omega_1\omega$ is also treated in Scott [65], Feferman [68] and Keisler [71], just to mention a few. We have chosen Karp[64] as our basis because it uses a Hilbert type proof theoretic approach to $L\omega_1\omega$.

2.1 The syntax of $L\omega_1\omega$

An $L\omega_1\omega$ *language* L is characterised 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*

$$F^n_0, F^n_1, \ldots, F^n_\xi, \ldots \qquad , \xi < \delta_2, \delta_2 < \omega_1$$

and the *n-place predicate symbols*

$$G^n_0, G^n_1, \ldots, G^n_\xi, \ldots \qquad , \xi < \delta_3, \delta_3 < \omega_1 .$$

Thus the language L can only have a countable number of non-logical symbols.

Each $L\omega_1\omega$ language L has the same set of *logical symbols*

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

and the same set of *variables*

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

Thus L has ω_1 variables.

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.

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

The other connectives and quantifiers are introduced as abbreviations as usual:

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

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

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

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

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

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 occurence 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 $1 \leq i,j \leq k$ and $i \neq j$, $x_i \neq x_j$. 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 ($\forall$vA') 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 some systematic fashion, so that the formula $A[t_1/x_1,\ldots,t_k/x_k]$ will be uniquely defined.

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

2.2 The semantics of $L\omega_1\omega$

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

A *structure for* L is a pair M = <D,I> , 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-assignments 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 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 = <D,I> 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 $(\bigwedge_{\xi<\delta} A_\xi)$, 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 $val_M(A_0,s<a_1/x_1,\ldots,a_k/x_k>)$ = tt for every $<a_1,\ldots,a_k> \in D^k$, 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 [64]. □

We say that the formula A *holds* in the structure M = <D,I>, 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 choice of V does not affect the property that a formula holds in M, as long as we choose a set V that contains each variable free in the formula.

We say that a structure M is a *model* for a set Δ of formulas, if each formula of Δ holds in M. The formula A is said to be a *semantic consequence* of the set Δ of formulas, 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 = <D,I> be a structure for L. A structure M' = <D,I'> 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[64] gives an axiom system for $L\omega_1\omega$. In this system we have the following *axiom schemes*:

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. $(\bigwedge_{\xi<\delta} (A_\delta \Rightarrow A_\xi) \Rightarrow (A_\delta \Rightarrow \bigwedge_{\xi<\delta} A_\xi))$, $0 < \delta < \omega_1$

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

Q1. $(\forall v(A_0 \Rightarrow A_1) \Rightarrow (A_0 \Rightarrow \forall vA_1))$, if no variable of v is free in A_0

Q2. $(\forall vA_0 \Rightarrow A_0[t_1/x_1,\ldots,t_k/x_k])$, where $x_1,\ldots,x_k$ are the distinct variables of v

E1. $t_1 = t_1$

E2. $(\bigwedge_{i\leq k} (t_i=t'_i) \Rightarrow F(t_1,\ldots,t_k) = F(t'_1,\ldots,t'_k))$

E3. $(\bigwedge_{i\leq k} (t_i=t'_i) \Rightarrow (G(t_1,\ldots,t_k) \Rightarrow G(t'_1,\ldots,t'_k)))$.

The *inference rules* of this axiom system are:

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

CN. $$\frac{A_0,\ \ldots,A_\xi,\ldots,\ \xi < \delta}{\bigwedge_{\xi<\delta} A_\xi}\ ,\qquad 0 < \delta < \omega_1$$

GN. $$\frac{A_0}{\forall vA_0}$$

Here $A_0,\ldots,A_\xi,\ldots$ are formulas of L, $t_1 \ldots,t_k,t'_1,\ldots t'_k$ are terms of L, F is a k-place function symbol of L, G is a k-place predicate symbol of L and v is a nonempty sequence of variables.

A *proof* in L of the formula A from the set Δ of formulas 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. We say that A is *provable from* Δ, denoted $\Delta \vdash A$, if there is a proof of A from Δ. We say that A is a *theorem*, if A is provable from the empty set of formulas.

The following results about $L\omega_1\omega$ will be useful later. The proof of these results is straightforward given the theorems proved in Karp [64]. We assume in the lemmas that Δ is a countable set of sentences of L.

LEMMA 2.2 (Completeness of $L\omega_1\omega$) For any formula A of L, $\Delta \vdash A$ if and only if $\Delta \models A$.

Proof: Follows from the theorems 11.2.4 and 11.4.1 in Karp [64]. □

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 [64]. □

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 [64]. □

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

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

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 $L\omega_1\omega$. The proof is here omitted. □

We will not give formal proofs of results in $L\omega_1\omega$ using the axioms, but will be content with 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 the proofs in $L\omega_1\omega$ may be infinitely long, a completely formal proof by exhibiting the sequence of formulas constituting the proof cannot in any case 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 the assumption formulas as constants. This means that we are not allowed to use the rule GN for universally quantifying variables that occur free in assumption formulas.

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 there is no reason to require program specifications to be deterministic.

The language will contain a new kind of primitive statement called an atomic description. This can be roughly 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, and also a nondeterministic binary choice construct.

The semantics of the descriptions will be of the denotational type, making use of the approximation relation for nondeterministic state transformations defined in Plotkin[76]. We will be following de Bakker[77a]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.

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 $\langle\rangle$. The set of elements in a list x is denoted $\tilde{x}$.

For any function $f{:}A \to B$, 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

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

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

$$\langle x/y\rangle = \langle x_1/y_1,\ldots,x_{\ell(x)}/y_{\ell(y)}\rangle \quad \text{and}$$

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

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* 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 formula of L, then

$\alpha x \beta y.Q$ *(atomic description)*

is a description. (The letters α and β are key-words, used to identify the lists x and y.)

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

(S ; S') (*composition*)

(S v S') (*nondeterministic choice*)

(B → S | S') (*selection*)

(B * S) (*iteration*)

are descriptions.

A program usually describes a state transformation, in which a set of variables are assigned new values. A description will be a generalisation of this, by also allowing the set of variables itself to be changed. Thus the effect of the atomic description $\alpha x \beta y.Q$, where $x = \langle x_1,\ldots,x_m \rangle$ and $y = \langle y_1,\ldots,y_n \rangle$, on a set V of variables is the following. The resulting set of variables will be $W = (V - \{y_1,\ldots,y_n\}) \cup \{x_1,\ldots,x_m\}$, i.e. the variable $y_1,\ldots,y_n$ will be deleted from V and the variables $x_1,\ldots,x_m$ will be added to V (some of the variables in x may belong to V already). The variables $x_1,\ldots,x_m$ are assigned new values so that the condition Q will be true, while all other variables of W have the same value they had before (Q is a condition on the variables in x and V). This transformation will be nondeterministic if there is more than one assignment of values to the variables in x that makes Q true, and it will be considered not to terminate when there is no assignment to x that makes Q true.

The descriptions (S ; S'), (B → S | S') and (B * S) provide the usual control structures (we write (B → S | S') for <u>if</u> B <u>then</u> S <u>else</u> S' and (B * S) for <u>while</u> B <u>do</u> S). The description (S v S') is a nondeterministic choice, i.e. either S or S' will be executed, the choice being made nondeterministically. Thus we have here an ordinary iterative language, with the exception of the atomic description. As arbitrary formulas are allowed in the atomic description, the descriptions are not usually machine executable.

Let fin(S,V) denote the resulting set of variables for a description S and an initial set V of variables. Then fin(S,V) is defined as follows. For every description S we have $\text{fin}(S,\emptyset) = \emptyset$. For a nonempty set V of variables we define fin(S,V) by cases as follows:

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

(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) $fin(B \rightarrow S'|S'',V) = \begin{cases} fin(S',V), & \text{if } var(B) \subset V, \ fin(S',V)=fin(S'',V) \\ \emptyset & \text{otherwise} \end{cases}$

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

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 $\rightarrow$ 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.

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 $\rightarrow$ W are determined as follows:

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

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

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

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

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. $R \subset D \times D$. 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 usually be omitted when it is clear from the context to which state space the undefined state belongs.

The idea of using the undefined state is to make the possibility of nontermination explicit. 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.

The set of all nonempty subsets of V_D will be denoted $P_D(V)$. Let W be another 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. If $f(s) \ni \bot$, then nontermination is also possible for the initial state s. 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 $f(\bot_{V,D}) = ff$. 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.

This way of defining state spaces, state transformations and state predicates is essentially the same as in de Bakker[77a], with the exceptions that we parameterize these with the initial and final spaces V and W, and that we do not require that the nondeterminism of the state transformations be bound. The first of these is motivated by our desire to treat state transformations in which the state space is altered, the second by the fact that we are interested in describing programs, and not only in the programs themselves.

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

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

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

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

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 $P_D(V)$, then U is said to *approximate* U', denoted $U \sqsubseteq U'$, if

either $U \ni \bot$ and $U - \{\bot\} \subset U'$

or $U \not\ni \bot$ 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 $F_D(V,W)$.

Proof: Omitted. □

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 $U \not\ni \bot$, 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 $U \ni \bot$, 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 $U - \{\bot\} \subset U'$. 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[77a] or Plotkin[76].)

The least element in $P_D(V)$ is element $\{\perp\}$ of $P_D(V)$. This follows from the fact that for any $U \in P_D(V)$, $\{\perp\} - \{\perp\} = \emptyset \subset U$, i.e. $\{\perp\} \sqsubseteq U$. 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 $f;g(s) \ni \perp$ for $s \in V_D$. Assume that $f;g(s) \neq \{\perp\}$ (otherwise we have directly that $f;g(s) \sqsubseteq f';g'(s)$), and let $s'' \in f;g(s)$, $s'' \neq \perp$. This means that for some $s' \in f(s)$, $s' \neq \perp$, $s'' \in g(s')$. 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 $f;g(s) \not\ni \perp$, then $f(s) \not\ni \perp$ and for any $s' \in f(s)$ we must have that $g(s') \not\ni \perp$.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 $(b \to f|g) \sqsubseteq (b \to f'|g')$, 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 for $s \in V_D$ with $b(s) = tt$ and $b(s) = ff$. □

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

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

if $U_n \ni \perp$ 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 $\perp$. 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

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

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 this information is not needed in the sequel.

Let $b \in E_D(V)$ and $f \in F_D(V,V)$. We define for $n \geq 0$ the transformations $(b * f)^n$ in $F_D(V,V)$, as follows:

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

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

We will prove that

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

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

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

Then the required result 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 $var(Q) \subset V$, 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.

A legal description $S:V \to W$, V and W nonempty sets of variables, will again 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) $$int_M(\alpha x\beta y.Q,V)(s) = \begin{cases} W(s), & \text{if } W(s) \neq \emptyset \\ \{\perp\}, & \text{if } W(s) = \emptyset . \end{cases}$$

Here $W(s) \subset W_D$ is defined for each $s \in V_D$, $s \neq \perp$, by

$W(s) \ni s'$ iff $val_M(Q,s\langle s'(x)/x\rangle) = tt$ and
$s(z) = s'(z)$ for each variable z in W, $z \notin \tilde{x}$.

(ii) $int_M(S';\, S'', V) = int_M(S',V)\ ;\ int_M(S'',fin(S',V))$,

$int_M(S' \vee S'', V) = int_M(S', V) \vee int_M(S'', V)$,

$int_M(B \to S'|S'', V) = (int_M(B, V) \to int_M(S', V) \mid int_M(S'', V))$,

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

Because $int_M(S,V)$ is a state transformation, we always have $int_M(S,V)(\perp) = \{\perp\}$. Note that for the atomic description, case (i), the final states do not depend on the values that the variables in x have in the initial state. For an intuitive understanding of the definition (i), we refer to page 17. The notation $s\langle s'(x)/x\rangle$ is explained by reference to pages 10 and 16: Assuming that x is the list $\langle x_1,\dots,x_m\rangle$, $s\langle s'(x)/x\rangle$ is an assignment of values to the variables in V and $\tilde{x}$, such that any variable z different from $x_1,\dots,x_m$ is assigned the value s(z), while the variable x_i is assigned the value $s'(x_i)$, for $i = 1,\dots,m$.

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 in turn 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, and it will be shown that the refinement relation captures the intuitive idea of a refinement step being correct. In this section we also show that the refinement relation is a preorder, thus justifying the stepwise manner of program construction.

In section 4.2 the weakest precondition of a state transformation is defined. It is shown that refinement between state transformation can be characterised 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

As remarked in the introduction, a refinement step is in an intuitive sense correct if it preserves the correctness of the program being refined. A more explicit formulation of this criteria can be given by introducing the notion of a program specification. We say that the refinement step leading from program S to program S' is correct, if the following condition holds:

> (A) For any program specification R, if S is totally correct with respect to R, then S' is totally correct with respect to R.

Programs will in this context be descriptions and will thus be interpreted as nondeterministic state transformations. Program specifications can also be interpreted as state transformations in the following way. Let the interpretation $M = \langle D,I \rangle$ be fixed, and let $S:V \to W$ be a description. A specification R of S must specify a set $U \subset V_D$ of initial states for which S must be guaranteed to terminate. It also has to specify for each $s \in U$ a set W_s of correct final states. This information can be expressed by the transformation $f_R \in F_D(V,W)$,

$$f_R(s) = \begin{cases} W_s, & \text{if } s \in U \\ \{\bot\}, & \text{if } s \notin U \end{cases} .$$

The description S will then be totally correct with respect to the specification R iff $f_R(s) \supset f(s)$ holds for each $s \in U$, where $f = int_M(S,V)$. Using the fact that $f_R(s) \not\ni \bot$ iff $s \in U$, an equivalent condition for S to be totally correct with respect to R is that $f_R(s) \not\ni \bot \Rightarrow f_R(s) \supset f(s)$, for any $s \in V_D$. This latter condition is taken as the definition of refinement between state transformations.

DEFINITION 4.1 (i) If U and U' are elements of $P_D(V)$, then U is *refined* by U', denoted $U \leq U'$, if $U \not\ni \bot \Rightarrow U \supset U'$.

(ii) If f and f' are elements of $F_D(V,W)$, then f is *refined* by f', denoted $f \leq f'$, if $f(s) \leq f'(s)$ for any $s \in V_D$.

Using the notation above, we thus have that S is totally correct with respect to R iff $f_R \leq f$ holds. Actually, any state transformation g can be considered as a specification, satisfied by S iff $g \leq f$ holds. This means that S must terminate for those initial states s for which $g(s) \not\ni \bot$, and produce a final state that belongs to the set g(s).

We are thus lead to the following characterisation of a correct refinement step. Let S and S' be two descriptions, S, S':V → W, with interpretations $f = int_M(S,V)$ and $f' = int_M(S',V)$. The refinement step leading from S to S' is then correct iff the following condition holds:

(A') For any $g \in F_D(V,W)$, if $g \leq f$ holds, then $g \leq f'$ holds.

LEMMA 4.1 The refinement relation is a preorder in $P_D(V)$ and $F_D(V,W)$.

Proof: We prove the lemma only for $P_D(V)$, from which the fact that refinement is a preorder in $F_D(V,W)$ then follows easily. For any U in $P_D(V)$ we have $U \supseteq U$, so $U \leq U$ will always hold, i.e. reflexivity is clear. To prove transitivity, assume that $U \leq U'$ and $U' \leq U''$ holds for U, U' and U'' in $P_D(V)$. If $U \ni \bot$, then $U \leq U''$ follows immediately. Otherwise, $U \supseteq U'$ must hold, and therefore $U' \not\ni \bot$, i.e. $U' \supseteq U''$. Thus $U \supseteq U''$ holds, i.e. $U \leq U''$ as desired. □

It turns out that the condition (A') holds iff $f \leq f'$ holds. This follows immediately from the fact that refinement is transitive and reflexive (from transitivity we get that $f \leq f'$ implies (A'), while reflexivity gives the converse). This gives us the final characterisation of a correct refinement step. Letting S, S', f and f' be as above, we have that the refinement step leading from S to S' is correct iff

(A'') $f \leq f'$ holds.

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.

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 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.

For deterministic programs, $S \leq S'$ reduces to the usual approximation relation between deterministic state transformations (see e.g. de Bakker[77a]), 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[74]), i.e. S and S' will terminate for the same initial states and will give the same final states for these initial states.

In Smythe[76] a relation similar to the refinement relation above is defined between state transformations of bounded nondeterminacy. Smythe uses it to prove the existence of a certain powerdomain construction in Plotkin[76], under weaker assumptions than those made by Plotkin. The refinement relation here has been arrived at independently of the work by Smythe, and is also used for an entirely different purpose. The connection between our work and the work by Smythe was pointed out by J. de Bakker.

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 $s \in V_D$, $s \neq \bot$,

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

As an immediate consequence of this definition, we see that if wp(f,q)(s) = tt for some $s \in V_D$, then $f(s) \not\ni \bot$, because $q(\bot) = ff$, 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[77a].

Using weakest preconditions, an alternative definition of total correctness for descriptions can be given. Let R and S be as in 4.1, i.e. R is a description from V to W that specifies the correct behaviour of the description S from V to W. R was interpreted as a state transformation f_R in $F_D(V,W)$, with

$$f_R(s) = W_s \quad \text{for each } s \in U, \text{ and}$$

$$f_R(s) = \{\bot\} \quad \text{for each } s \notin U.$$

Here U was the set of initial states for which S was required to terminate, while W_s was the set of correct final states of S for each initial state $s \in U$, $W_s \not\ni \bot$.

Let the interpretation of S in $F_D(V,W)$ be f, and define for each $s \in U$ a state predicate q_s in $E_D(W)$, by

$$q_s(s') = tt \text{ iff } s' \in W_s.$$

Then we have that S will be correct with respect to R iff for each $s \in V_D$,

$$\text{if } s \in U, \text{ then } wp(f\ ,q_s)(s) = tt. \qquad (1)$$

To see that this is really the case, assume first that condition (1) holds. If $s \in U$, then $wp(f\ ,q_s)(s) = tt$, i.e. for any $s' \in f(s)$, $q_s(s') = tt$, and thus $s' \in W_s$ for any $s' \in f(s)$. On the other hand, if $s' \in W_s$ for any $s' \in f(s)$, $s \in U$, then $q_s(s') = tt$ for any $s' \in f(s)$ and so $wp(f\ ,q_s)(s) = tt$.

The boolean operations on truth values can be extended to state predicates as follows. Let p and p' be two state predicates in $E_D(V)$. Then $(p \wedge p')$ is a state predicate in $E_D(V)$, defined by

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

Similarly for the boolean connectives $\sim$, $\vee$, $\Rightarrow$, $\Leftrightarrow$. It will be convienient to use the predicate p also as expressing the condition $p(s) = tt$ for each $s \in V_D$. This is done in the following theorem and will also be used later.

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

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

for any state predicate q in $E_D(W)$.

Proof: ($\Rightarrow$) Assume that $f \leq f'$ and let q be any predicate in $E_D(W)$. Let $s \in V_D$ be such that $wp(f,q)(s) = tt$. This means that $f(s) \not\ni \perp$, and using the assumption, this means that $f(s) \supset f'(s)$. 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 $f(s) \not\ni \perp$. 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 $f'(s) \subset f(s)$. This shows that $f \leq f'$. □

This theorem will be used in the next chapter to give a technique for proving refinement between descriptions.

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 then derived, and its soundness and completeness proved. This proof rule makes use of the weakest preconditions of descriptions, and is based on theorem 4.3 above.

In section 5.2 we will also give a proof rule for equivalence of descriptions and present a very useful induction rule for iteration, together with some other properties of refinement. In section 5.3 we generalise somewhat the properties of weakest preconditions given in Dijkstra[76]. These properties will be needed in section 5.4 and later. In section 5.4 we will finally prove an important replacement property of descriptions, which will provide a justification for the top-down program development strategy, further discussed in the next chapter.

5.1 Weakest preconditions of descriptions

Let $S:V \to W$ be a description and Q a formula of L, $var(Q) \subset W$. 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, $var(P) \subset V$, that describes the weakest precondition $wp(f,q)$ of f and q, i.e. we require that

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

This formula P will then give the weakest condition that an initial state must satisfy so that the execution of S is guaranteed to terminate, and so that any final state of S will satisfy the 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 has the property (5.1) required.

We introduce the abbreviations true and false for sentences of $L\omega_1\omega$ by

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

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

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

$$skip =_{df} \alpha\langle\rangle\beta\langle\rangle.true \quad \text{and}$$

$$abort =_{df} \alpha\langle\rangle\beta\langle\rangle.false \ .$$

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

$$int_M(skip,V) = \Lambda_{V,D}$$

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

$$int_M(abort,V) = \Omega_{V,D} \ .$$

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

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

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

Using induction on n, it is easily verified that

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

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, $var(R) \subset W$. Then the *weakest precondition* of S for R, denoted WP(S,R), is defined by induction on the structure of S, as follows:

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

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

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

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

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

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

$$WP(\text{skip},\ R) = \text{true} \wedge (\text{true} \Rightarrow R) \Leftrightarrow R, \text{ and}$$

$$WP(\text{abort}, R) = \text{false} \wedge (\text{false} \Rightarrow R) \Leftrightarrow \text{false},$$

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, $var(R) \subset W$, then WP(S, R) is a formula of L, with $var(WP(S,R)) \subset V$.

Proof: The proof goes by induction on the structure of S. We show here only the basis step, i.e. the case when $S = \alpha x\beta y.Q$. Because $var(Q) \subset V \cup \tilde{x}$, we have $var(\exists xQ) \subset V$, as no variable in x is free in $\exists xQ$. Also, because $var(R) \subset W$, and $W = (V - \tilde{y}) \cup \tilde{x} \subset V \cup \tilde{x}$, we have $var(\forall x(Q \Rightarrow R)) \subset var(Q) \cup var(R) - \tilde{x}$ i.e. $var(\forall x(Q \Rightarrow R)) \subset V$. This means that $var(WP(\alpha x\beta y.Q,R) \subset V$. The induction step, i.e. cases (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 ≠ ∅, and let R be a formula of L, var(R) ⊂ W. Then, for any structure M of L, we have that

$$\text{int}_M(\text{WP}(S, R), V) = \text{wp}(\text{int}_M(S,V), \text{int}_M(R,W)).$$

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

$$f = \text{int}_M(S,V) \in F_D(V,W) \text{ and}$$

$$r = \text{int}_M(R,W) \in E_D(W).$$

We have then to prove that

$$\text{int}_M(\text{WP}(S,R),V) = \text{wp}(f,r).$$

(i) S is $\alpha x\beta y.Q$. We have in this case that f(s) = W(s), if W(s) ≠ ∅, and f(s) = {⊥}, if W(s) = ∅, where W(s) is defined by

$$W(s) \ni s' \quad \text{iff} \quad \text{val}_M(Q, s\langle s'(x)/x\rangle) = tt \text{ and}$$
$$s(z) = s'(z) \text{ for each } z \in W - \tilde{x},$$

by the definition of the semantics of the atomic description.

(⇒) Let $s \in V_D$ such that $\text{int}_M(\text{WP}(S,R),V)(s) = tt$. This means that

$$\text{val}_M(\exists xQ, s) = tt \text{ and } \text{val}_M(\forall x(Q \Rightarrow R), s) = tt,$$

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

Now $\text{val}_M(\exists xQ, s) = tt$ iff $\text{val}_M(Q, s\langle d/x\rangle) = tt$ 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 $\text{val}_M(Q, s\langle s'(x)/x\rangle) = tt$, i.e. $s' \in W(s)$. Therefore W(s) ≠ ∅, and we have that f(s) = W(s).

Assume now that $s' \in W(s)$, which implies that $s \neq \bot$. Then $s'(z) = s(z)$ for $z \in W - \tilde{x}$, and $val_M(Q, s<s'(x)/x>) = tt$. By assumption, $val_M(\forall x(Q \Rightarrow R),s) = tt$, i.e. $val_M(Q \Rightarrow R , s<d/x>) = tt$ for any list d of elements in D. This means that $val_M(R, s<s'(x)/x>) = tt$, by choosing $d = s'(x)$ and using modus ponens. Because $s<s'(x)/x> (z) = s'(z)$ for any $z \in W$, and $var(R) \subset W$, 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 $\bot \notin f(s)$, because $r(\bot) = ff$. Thus $W(s) \neq \emptyset$, i.e. there is an $s' \in W_D$ such that $val_M(Q, s<s'(x)/x>) = tt$ and $s'(z) = s(z)$ for $z \in W - \tilde{x}$. Thus $val_M(\exists xQ,s) = tt$.

Assume that $val_M(Q, s<d/x>) = tt$. Define $s' \in W_D$ by $s'(z) = s(z)$ for $z \in W - \tilde{x}$, and $s'(x_i) = d_i$ for $i = 1,\dots,\ell(x)$. Then $s' \in f(s)$, which implies that $r(s') = tt$, i.e. $val_M(R, s') = tt$. Because $var(R) \subset W$ and $s<d/x>(z) = s'(z)$ for $z \in W$, we have from this that $val_M(R, s<d/x>) = tt$. This gives $val_M(Q \Rightarrow R, s<d/x>) = tt$, 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

$$\begin{aligned} int_M(WP(S';S'',R),V) &= int_M(WP(S',WP(S'',R)),V) \\ &= wp(f', int_M(WP(S'',R),V')) \quad \text{(by induction hyp.)} \\ &= wp(f', wp(f'',r)) \quad \text{(by the induction hyp. again).} \end{aligned}$$

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 prove first that for each $n < \omega$,

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

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 $n \geq 0$.

$$\begin{aligned} & int_M(WP((B * S')^{n+1},R),V) \\ & \quad = int_M(WP((B \to S';(B * S')^n \mid skip), R),V) \\ & \quad = int_M((B \Rightarrow WP(S';(B * S')^n,R)) \wedge (\sim B \Rightarrow R), V) \\ & \quad = (b \Rightarrow wp(f', int_M(WP((B * S')^n,R),V)) \wedge (\sim b \Rightarrow r) \\ & \quad = (b \Rightarrow wp(f', wp((b * f')^n,r))) \wedge (\sim b \Rightarrow r) \text{ (ind. hyp)} \\ & \quad = wp((b \Rightarrow f';(b * f')^n \mid \Lambda_{V,D}), r) \\ & \quad = wp((b * f')^{n+1},r). \end{aligned}$$

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

To prove the 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 $(b * f')^n(s) \not\ni \bot$, 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 $(b * f')^n(s) \ni \bot$ for every $n < \omega$. In this case we have that

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

and thus $(b * f')(s) \ni \bot$. Therefore $wp(b * f',r)(s) = ff$. If on the other hand, $(b * f')^n(s) \not\ni \bot$ 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 get the conclusion 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, $(S' \vee S'')$ and $(B \rightarrow S' | S'')$, are omitted. □

A similar theorem is proved in de Bakker[77b]. 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. (The use of $L\omega_1\omega$ in connection with program correctness is also advocated in Engeler[75].)

5.2 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 $M = \langle D,I \rangle$ be a structure for L. We have by definition 4.2, that $S \leq_M S'$ iff

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

By theorem 4.3 we have that this holds iff

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

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

$$wp(int_{M'}(S,V),int_{M'}(G(w),W)) \Rightarrow wp(int_{M'}(S',V),int_{M'}(G(w),W))$$
$$\text{for any expansion } M' \text{ of } M \text{ to } L'.$$

We have here used the fact that $int_{M'}(S,V) = int_M(S,V)$ and the same for S', because 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

$$int_{M'}(WP(S,G(w)),V) \Rightarrow int_{M'}(WP(S',G(w)),V), \text{ for any expansion}$$
$$M' \text{ of } M \text{ to } L'.$$

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

$$S \leq_M S' \quad \text{for any model } M \text{ of } \Delta .$$

This is again by theorem 5.3 the case iff

$$WP(S,G(w)) \Rightarrow WP(S',G(w)) \text{ holds in any expansion } M' \text{ of } M \text{ to } L, \text{ for any model } M \text{ of } \Delta . \qquad (5.4)$$

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

$$WP(S,G(w)) \Rightarrow WP(S',G(w)) \text{ holds in any model } M' \text{ of } \Delta, M' \text{ a structure for } L'. \qquad (5.5)$$

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

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

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 say 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 Δ of sentences, 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. On the other hand, the proof technique is *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 theorems 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, $var(Q) \subset W$. If $S \leq_M S'$, then

$$WP(S,Q) \Rightarrow WP(S',Q) \text{ holds in } M.$$

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

$$WP(S,Q) \Rightarrow WP(S',Q) \quad \text{holds in } M. \quad \square$$

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. $\square$

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 cor. 5.5. $\square$

Finally, we prove a simple induction rule for iteration that 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, $var(B) \subset V$. 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

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

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

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

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

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

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

Thus we have that

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

giving the required result

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

5.3 Basic properties of weakest preconditions

Dijkstra[76] gives five basic properties of weakest preconditions for 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 would be the following ($var(Q)$, $var(Q') \subset W$):

(1) $WP(S, \text{false}) \Leftrightarrow \text{false}$

(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, $var(Q_i) \subset W$ for $i < \omega$, then

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

The first four of these properties will hold for any descriptions S, while property (5) will only hold for descriptions S when their nondeterminacy is *bounded*. This means that the interpretation of S in the structure $M = \langle D,I\rangle$ must satisfy the condition: for any $s \in V_D$, either $int_M(S,V)(s)$ is a finite set, or then $int_M(S,V)(s)$ contains the undefined state $\perp$. The counterexample that is used in Dijkstra[76] for showing that (5) does not necessarily hold when the nondeterminacy is not bound can be used also for showing that property (5) does not necessarily hold for descriptions (the counterexample used by Dijkstra can be expressed as a description but not as a guarded command).

Before giving the appropriate generalisations of the first four properties, we need to make a preliminary definition (used only 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 $\alpha x \beta y.Q$, then z does not belong to $\tilde{x}$, and

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

LEMMA 5.11 Let $S:V \rightarrow W$ be a legal description. Let Q_n be formulas of L, $var(Q_n) \subset W$, for $n < \omega$. Further, let x be a list of distinct variables, such that any variable in $W - \tilde{x}$ is constant in S. Then

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

(ii) $\forall x(Q_0 \Rightarrow Q_1) \Rightarrow (WP(S,Q_0) \Rightarrow WP(S,Q_1))$,

(iii) $WP(S, \bigwedge_{n<\alpha} Q_n) \Leftrightarrow \bigwedge_{n<\alpha} WP(S, Q_n)$, $\alpha < \omega_1$

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

hold in any structure M of L.

Proof: The proofs of all these cases are quite similar and are all based on theorem 5.2. We will prove case (ii) as an example.

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 $int_M(S,V)(s) \ni s'$, 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), $val_M(Q_0, s<d/x>) = tt$ implies $val_M(Q_1, s<d/x>) = tt$ for any list d of elements in D, $\ell(d) = \ell(x)$. Because $var(Q_0) \subset W$ and $s(z) = s'(z)$ for $z \in W - \tilde{x}$, we have that $val_M(Q_0,s') = val_M(Q_0, s<s'(x)/x>) = tt$, giving $val_M(Q_1, s<s'(x)/x>) = tt$, 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. □

Note that formulas (i) - (iv) will be provable from any countable set Δ of sentences of L, as a consequence of lemma 5.11 and the completeness of $L\omega_1\omega$.

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

$$\Delta \vdash S_1 \leq S_1' \quad \text{and} \tag{5.6}$$

$$\Delta \vdash S_2 \leq S_2' \ . \tag{5.7}$$

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 $L\omega_1\omega$ (subchapter 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(ii), 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

$$\Delta \vdash WP(S_1, WP(S_2,G(v))) \Rightarrow WP(S_1', WP(S_2',G(v))), \text{ i.e.}$$

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

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

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

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

The analoguous 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, $\mathrm{var}(B) \subset V$. Assume that

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

We first show that

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

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

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

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 first 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

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

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 summarise these results in the following theorem.

THEOREM 5.12 (Replacement) Let $S:V \rightarrow W$ be a legal description, containing the subdescription $T:V' \rightarrow W'$. Let $T':V' \rightarrow W'$ be a legal description, and let $S':V \rightarrow W$ 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

$$\Delta \vdash T \leq T' \quad \text{implies} \quad \Delta \vdash S \leq S'.$$

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

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[76], 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 a restricted form of descriptions called *program descriptions*, which are better suited for program development. We will compute the weakest preconditions for the program descriptions using the rules for computing weakest preconditions for descriptions. Programs will finally be special kinds of program descriptions, and will in effect be the guarded commands of Dijkstra[76].

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[76] , 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 formalisation 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

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

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

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

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

S_1: h,z:= X^Y,1; {P has been established}
<u>do</u> h ≠ 1 → squeeze h under invariance of P <u>od</u>
{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 <u>do</u> h ≠ 1 → ... <u>od</u> construction is a loop; the statement ... is repeated as long as the condition h ≠ 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. 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

$$h = x^y.$$

In stead 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:

S_2: x,y,z:= X,Y,1; {P has been established}
<u>do</u> y ≠ 0 → y,z:= y-1,z•x {P has not been destroyed} <u>od</u>
{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

<u>do</u> 2|y → x,y:= x•x,y/2 <u>od</u>

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,
```

which gives 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 optimising transformation).

The application of both top-down development and optimising 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 a part of the program when making a refinement step. Optimising transformations are again useful in removing inefficiencies introduced by the top-down approach when the interaction between different program parts was not considered.

6.2 Correct refinements using descriptions

In this section we will discuss principles for developing programs so 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 exhibited in the preceeding section, while still staying in the framework of refinement between descriptions developed in the preceding chapter.

1. TOP-DOWN DEVELOPMENT. The fact that the transitivity of refinement justifies a stepwise construction of the final program was noted already 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, ..., 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 correct.

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, however, easily achieved. Consider e.g. the assignment statement

$$x := x+y.$$

The same effect can be achieved with the description

$$\alpha<z>\beta<>.\ z = x+y;$$
$$\alpha<x>\beta<z>.\ x = z \quad ,$$

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

$$x := x/y$$

would again be expressed by the description

$$\alpha<z>\beta<>.\ z = x/y \wedge y \neq 0;$$
$$\alpha<x>\beta<z>.\ x = z \quad .$$

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

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

$$\alpha<x>\beta<>.\ x = x+y,$$

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. Thus we here have an example of a perfectly acceptable 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 <u>if</u> ... <u>fi</u> and the <u>do</u> ... <u>od</u> constructions are expressible using descriptions, i.e. the programs of the previous section can be expressed as descriptions.

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 simpler 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

$$\alpha\langle\rangle\beta\langle\rangle.\ R\ ,$$

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.

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

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

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

If we then can prove that

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

we have by theorem 5.12 that

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

where

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

Transitivity then gives the desired result, i.e.

$$S \leq S''.$$

Thinking operationally,

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

states that formula R will be invariantly true at the indicated place of the description when the execution starts in an initial state for which S is guaranteed to terminate. To see this it is enough to notice that if this was not true, then for some initial state for which S was guaranteed to terminate, it would be possible for S' not to terminate. This would then contradict the assumption that $S \leq S'$ holds.

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 program 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 $L\omega_1\omega$. The correctness of a program transformation rule

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

where Φ is the set of assumptions made, can be shown by deriving $S \leq \tau(S)$ in $L\omega_1\omega$ from the assumptions Φ . In chapter 7 program transformation rules of this kind will be treated extensively and their correctness shown in the manner suggested. These program transformations will be concerned with the introduction of assertions into descriptions (section 7.3), the use of representational abstraction (section 7.4) and changing the control structure in a description (section 7.5).

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

$$\underline{\text{set}}<x>.\ |x^2 - x'| < e.$$

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

$$\alpha<z>\beta<>.\ |x^2 - z| < e\ ;$$
$$\alpha<x>\beta<z>.\ x = z\ ,$$

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' as before 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 from changing e). The fact that the description S satisfies these entry and exit conditions can be expressed by

$$\{e > 0\};\ \underline{\text{set}}\langle x\rangle.|x^2 - x'| < e \quad \underline{<} \quad S \quad . \qquad (6.1)$$

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

Operational abstraction can be achieved by using the procedure specification

$$\{e > 0\};\ \underline{\text{set}}\langle x\rangle.|x^2 - x'| < e$$

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. This of course makes it impossible to use recursive procedures.

In section 7.2 of the next chapter we 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. REPRESENTATIONAL ABSTRACTION. An example of representational abstraction was already provided in the preceding section, in the transition from program S_1 to program S_2. Another example is the following.

Consider a program S using a set V of variables. Let a be one of the variables of S, taking only small sets of integers as values during the execution of S (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. that 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 small set of integers represented by b and k. In this case we give

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

Here the function t is the *abstraction function* and the condition I the *concrete invariant* introduced in Hoare[72] 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 than 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 $C: W \rightarrow V$, defined by

$$C = \alpha\langle a\rangle\beta\langle b,k\rangle.\ a = t(b,k) \wedge I(b,k).$$

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 $D: V \rightarrow W$, defined by

$$D = \alpha\langle b,k\rangle\beta\langle a\rangle.\ a = t(b,k) \wedge I(b,k).$$

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 C and D are each others inverses. Note that description C is deterministic while description D 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.

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 $S': W \rightarrow W$ such that

$$\{R\};\ S \leq D;\ S';\ C \qquad (6.2)$$

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 D 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 by the following recursive procedure. We may always simply invent an S' satisfying (6.2), and then the problem is solved. If, however, S is of the form $(S_1;S_2)$, $(S_1 \vee S_2)$, $(B \rightarrow S_1 \mid S_2)$ or $(B * S_1)$, where $S_1,S_2:V \rightarrow V$, there is another possibility open. 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 subproblem of finding S_1' and S_2' that satisfy

$$\{R\};S_1 \leq D;\ S_1';\ C \quad \text{and}$$
$$\{R\};S_2 \leq D;\ S_2';\ C.$$

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

$$\{R\};S_1;\{R\};S_2 \leq (D;S_1';C);(D;S_2';C).$$

Finally, it can be shown that the transformation rule

$$(D;S_1';C);(D;S_2';C) \leq D;(S_1';S_2');C \qquad (6.3)$$

is always correct, provided C and D satisfy certain properties (which they do in this example). Transitivity of refinement then gives us the desired result,i.e

$$\{R\};S \leq D;S';C\ ,$$

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 7.4 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 initialise the variable a to some value, and it does not depend on the initial value of a. In this case we introduce the description $D_0:V \to W$, defined by

$$D_0 = \alpha\langle b,k\rangle\beta\langle a\rangle.\ \text{true}.$$

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

$$S \leq D_0;\ S';\ C$$

holds. The restriction R can be dropped here because D_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

$$S_1 \leq D_0;\ S_1';\ C \quad \text{and}$$

$$\{R\};S_2 \leq D;\ S_2';\ C\ .$$

By replacement we again get that

$$S_1;\ \{R\};\ S_2 \leq (D_0;\ S_1';\ C);\ (D;\ S_2';\ C).$$

Finally we use a program transformation rule that gives

$$(D_0;\ S_1';\ C);\ (D;\ S_2';\ C) \leq D_0;\ (S_1';\ S_2');\ C\ .$$

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

$$S \leq D_0;\ S';\ C,$$

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

If the final value of a does not matter either, i.e. any final value of a is allowed, then we can also introduce the description $C_0 : W \to V$, defined by

$$C_0 = \alpha\langle a\rangle\beta\langle b,k\rangle.\ \text{true}\ ,$$

and consider the problem of finding a description $S' : W \to W$ satisfying

$$S \le D_0;\ S';\ C_0,$$

which can again be solved by the same technique.

The approach to stepwise refinement presented above is new, as far as we know. Related ideas have, however, been presented before. Thus Katz & Manna[76] contains a similar technique of using assertions to collect information about the context of a program part. The nondeterministic assignment has been used previously by Harel & al[77] in the extension they give of Hoare'a axiomatic system. The formalisation of representational abstraction given here is clearly inspired by the *abstract data type* facility first discussed in Hoare[72a], and provided in a number of new programming languages (see e.g. Wulf & al[77], Wirth[77], Lampson & al[77] and Liskov & al[77]). 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 & Darlington[75] or the concept of simulation between programs defined in Milner[71]

6.3 Program descriptions

In this section a special kind of descriptions called *program descriptions* will be defined. Program development along the lines discussed in the previous section is intended to be carried out using only this kind of descriptions. A special notation is introduced for the program descriptions, to make their use more convienient. *Programs* and *program specifications* will again be special kinds of program descriptions.

Because the program descriptions, programs and program specifications all are special kinds of descriptions, it is possible to compute the weakest preconditions for these constructs using the rules for computing the weakest preconditions for descriptions. We will do this below, at the same time as we define the set of 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.

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

$$\{Q\} =_{df} \alpha\langle\rangle\beta\langle\rangle.Q$$

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

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

and *abort statement*

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

The skip and the abort statement have here the same meaning as they have in Dijkstra[76].

The weakest preconditions for these constructs are as follows:

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

$$WP(\text{skip}, R) \Leftrightarrow R,$$

$$WP(\text{abort}, R) \Leftrightarrow \text{false}.$$

This follows directly by computation. We have

$$\begin{aligned} WP(\{Q\}, R) &= WP(\alpha\langle\rangle\beta\langle\rangle.Q, R) \\ &\Leftrightarrow Q \wedge (Q \Rightarrow R) \\ &\Leftrightarrow Q \wedge R. \end{aligned}$$

We then have that

$$WP(\text{skip}, R) \Leftrightarrow \text{true} \wedge R \Leftrightarrow R \text{ and}$$

$$WP(\text{abort}, R) \Leftrightarrow \text{false} \wedge R \Leftrightarrow \text{false}.$$

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

2. *ASSIGNMENTS*. Let Q be a formula of L, and let x be a list of distinct variables in V, where $var(Q) \subset V \cup \tilde{x}$. Then the *(nondeterministic) assignment*

$$\underline{set}\ x.Q =_{df} \alpha x'\beta\langle\rangle.Q;\ \alpha x\beta x'.x=x'$$

is a program description in V. A special case of the assignment is the *assignment statement*

$$x := t =_{df} \underline{set}\ x.\ x' = t,$$

where x is a list of distinct variables of V and t is a list of terms of L, $\ell(x) = \ell(t)$ and $var(t_i) \subset V$ for $i = 1,\ldots,\ell(t)$. The angular brackets will usually be omitted in connection with the assignment statement,in examples, i.e. we will write $x_1,\ldots,x_{\ell(x)} := t_1,\ldots,t_{\ell(t)}$ is stead of the more correct $\langle x_1,\ldots,x_{\ell(x)}\rangle := \langle t_1,\ldots,t_{\ell(t)}\rangle$.

The effect of the assignment is to assign new values to the variables in the list x, so that condition Q will be true. In Q the marked variables x' stand for the new values assigned to the variables x, while the program variables x stand for the old values. No other variable is affected by the execution of the the assignment.

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

$$WP(\underline{set}\ x.Q,\ R) \Leftrightarrow \exists x'Q \wedge \forall x'(Q \Rightarrow R[x'/x]) \quad \text{and}$$

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

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

$$WP(\underline{set}\ x.Q,\ R) = WP(\alpha x'\beta\langle\rangle.Q,\ WP(\alpha x\beta x'.x=x',\ R)).$$

We have

$$\begin{aligned} WP(\alpha x\beta x'.x=x',\ R) &= \exists x(x=x') \wedge \forall x(x=x' \Rightarrow R) \\ &\Leftrightarrow true \wedge R[x'/x] \text{ (by lemma 2.6)} \\ &\Leftrightarrow R[x'/x]. \end{aligned}$$

Thus

$$WP(\underline{set}\ x.Q,\ R) \Leftrightarrow WP(\alpha x'\beta<>.Q,\ R[x'/x])$$
$$\Leftrightarrow \exists x'Q \wedge \forall x'(Q \Rightarrow R[x'/x]).$$

For the assignment statement we have

$$WP(x := t,\ R) = WP(\underline{set}\ x.\ x' = t,\ R)$$
$$\Leftrightarrow \exists x'(x' = t) \wedge \forall x'(x' = t \Rightarrow R[x'/x])$$
$$\Leftrightarrow true \wedge R[t/x]$$
$$\Leftrightarrow R[t/x].$$

3. ABSTRACTION. Let $\alpha x\beta y.Q$ 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, $var(t_i) \subset W$ for $i = 1,\ldots,\ell(t)$, $var(I) \subset W$ and $\ell(t) = \ell(y)$. Let S be a program description in W. Then the *abstractions*

$$\underline{rep}\ \alpha x\beta y.Q\colon\ S\ \underline{per} =_{df} \alpha x\beta y.Q;S;\alpha y\beta x.Q,$$
$$\underline{rep}\ \alpha x\beta y.Q\colon\ S\ \underline{end} =_{df} \alpha x\beta y.Q;S;\alpha y\beta x.true$$
$$\underline{beg}\ \alpha x\beta y.Q\colon\ S\ \underline{per} =_{df} \alpha x\beta y.true;S;\alpha y\beta x.Q \quad \text{and}$$
$$\underline{beg}\ \alpha x\beta y\colon\ S\ \underline{end} =_{df} \alpha x\beta y.true;S;\alpha y\beta x.true.$$

are program descriptions in V.

The weakest preconditions for these constructions are:

$$WP(\underline{rep}\ \alpha x\beta y.Q\colon\ S\ \underline{per},\ R)$$
$$\Leftrightarrow \exists x(\ y{=}t \wedge I) \wedge$$
$$\forall x(\ y{=}t \wedge I \Rightarrow WP(S,\ I \wedge R[t/y])),$$

$$WP(\underline{rep}\ \alpha x\beta y.Q\colon\ S\ \underline{end},\ R)$$
$$\Leftrightarrow \exists x(y{=}t \wedge I) \wedge$$
$$\forall x(y{=}t \wedge I \Rightarrow WP(S,\ \forall yR)),$$

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

$$WP(\underline{beg}\ \alpha x\beta y\colon S\ \underline{end},\ R) \Leftrightarrow \forall x WP(S,\ \forall y R).$$

The computation of these weakest preconditions goes as follows. We compute first

$$WP(\alpha y\beta x.y{=}t \wedge I, R) = \exists y(y{=}t \wedge I) \wedge \forall y(y{=}t \wedge I \Rightarrow R).$$

We have by lemma 2.6 that

$$\exists y(y{=}t \wedge I) \Leftrightarrow I[t/y] \Leftrightarrow I,$$

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

$$\forall y(y{=}t \wedge I \Rightarrow R) \Leftrightarrow \forall y(I \Rightarrow (y{=}t \Rightarrow R)) \Leftrightarrow I \Rightarrow \forall y(y{=}t \Rightarrow R)$$

$$\Leftrightarrow I \Rightarrow R[t/y],$$

for the same reason. Thus we get that

$$WP(\alpha y\beta x.y{=}t \wedge I,\ R) \Leftrightarrow I \wedge (I \Rightarrow R[t/y]) \Leftrightarrow I \wedge R[t/y].$$

We also get that

$$WP(\alpha y\beta x.\text{true},\ R) = \exists y(\text{true}) \wedge \forall y(\text{true} \Rightarrow R) \Leftrightarrow \forall y R.$$

Thus the result will follow by computing

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

$$\Leftrightarrow WP(\alpha x\beta y.y{=}t \wedge I,\ WP(S,\ I \wedge R[t/y])),$$

$$WP(\underline{rep}\ \alpha x\beta y.y{=}t \wedge I\colon S\ \underline{end},\ R)$$

$$\Leftrightarrow WP(\alpha x\beta y.y{=}t \wedge I,\ WP(S,\ \forall y R)) \quad \text{and}$$

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

$$\Leftrightarrow WP(\alpha x\beta y.\text{true},\ WP(S,\ I \wedge R[t/y]))$$

$$WP(\underline{beg}\ \alpha x\beta y\colon S\ \underline{end},\ R)$$

$$\Leftrightarrow WP(\alpha x\beta y.\text{true},\ WP(S,\ \forall y R)).$$

The purpose of an abstraction is to allow a change of state space to take place temporarily. The abstraction

$\underline{\text{rep}}\ \alpha x \beta y.y=t \wedge I\text{: } S\ \underline{\text{per}}$

will achieve this change by replacing the variables y in V with new variables x that represent the values of the variables in y by the equation

$y_i = t_i$, 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 the values of the variables in y. The values chosen for x must, however, satisfy the condition I.

After the variables in y have been replaced with the variables in x, the description S is executed and the variables in y are then assigned the values represented by the new values computed by S for x (and for the other variables in W). All in all, the effect of the abstraction is to manipulate the variables in y by manipulating a representation of these variables.

The abstraction

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

is used to initialise the values of the variables in y by initialising the values of the variables in x used to represent the variables in y. The abstraction

$\underline{\text{rep}}\ \alpha x \beta y.y=t \wedge I\text{: } S\ \underline{\text{end}}$

is again used in cases where the representation of y by x may be destroyed.

The last abstraction

$\underline{\text{beg}}\ \alpha x \beta y\text{: } S\ \underline{\text{end}}$

is used for introducing new temporary variables at the same time as some other variables are deleted. The deleted variables will, however, not get their old values after this description has been performed, but will be assigned some arbitrary values. A special case of this last abstraction is the *block*

$$\underline{\text{beg}}\ x\text{: } S\ \underline{\text{end}} \ =_{df}\ \underline{\text{beg}}\ \alpha x\beta\langle\rangle\text{: } S\ \underline{\text{end}},$$

where x must be a list of program variables not appearing in V and S is a program description in $V \cup \tilde{x}$. The weakest precondition for the block will be

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

which is easily verified by noting that $\forall yR$ is by definition R when $y = \langle\rangle$.

4. 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 $(S_1;(S_2;(\ldots;(S_{n-1};S_n)\ldots)))$.

5. NONDETERMINISTIC SELECTION Let $S_1, \ldots, S_n$ be program descriptions in V, and let $B_1,\ldots,B_n$ be formulas of L, such that $\text{var}(B_i) \subset V$ for $i = 1,\ldots,n$, $n \geq 1$. The *nondeterministic selection*

$$\underline{\text{if}}\ B_1 \rightarrow S_1 \mid \ldots \mid B_n \rightarrow S_n\ \underline{\text{fi}}$$

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

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

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

$$\underline{\text{if}}\ B_1 \rightarrow S_1 \mid \ldots \mid B_n \rightarrow S_n\ \underline{\text{fi}}$$
$$= \underline{\text{if}}\ B_1 \rightarrow S_1 \mid B_2 \vee \ldots \vee B_n \rightarrow \underline{\text{if}}\ B_2 \rightarrow S_2 \mid \ldots \mid B_n \rightarrow S_n\ \underline{\text{fi}}\ \ \underline{\text{fi}},$$

for $n > 2$.

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

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

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

6. *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 $\mathrm{var}(B_i) \subset V$ for $i = 1,\ldots,n$, $n \ge 1$. Then the *nondeterministic iteration*

$$\underline{\text{do}}\ B_1 \to S_1 \mid \ldots \mid B_n \to S_n\ \underline{\text{od}}$$

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

is a program description in V.

The weakest precondition for nondeterministic iteration is

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

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

where

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

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

for $n \ge 0$.

Noting that

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

we find that

$$\underline{\text{do}}\ B_1 \to S_1 \mid \ldots \mid B_n \to S_n\ \underline{\text{od}}^0 = \text{abort} \quad \text{and}$$

$$\underline{\text{do}}\ B_1 \rightarrow S_1 \mid \dots \mid B_n \rightarrow S_n\ \underline{\text{od}}^n$$
$$\approx \underline{\text{if}}\ BB \rightarrow \underline{\text{if}}\ B_1 \rightarrow S_1 \mid \dots \mid B_n \rightarrow S_n\ \underline{\text{fi}};$$
$$\underline{\text{do}}\ B_1 \rightarrow S_1 \mid \dots \mid B_n \rightarrow S_n\ \underline{\text{od}}^{n-1}$$
$$\mid \sim BB \rightarrow \text{skip}$$
$$\underline{\text{fi}}, \quad \text{for } n > 0,$$

where we denote with BB the condition $B_1 \vee \dots \vee B_n$.

The program descriptions are now the descriptions generated by the rules (1) to (6) above. The *programs* are generated by these same rules, when restricted as follows: in (1) we only allow the skip and abort statement, in (2) only the assignment statement, in (3) only the block, (4) is unrestricted and in (5) and (6) the formulas $B_1,\dots,B_n$ may not contain any quantifiers or infinite disjunctions or conjunctions.Thus the programs are the *guarded commands* of Dijkstra[76], plus the block construction. The weakest preconditions for the 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 will finally be 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}; <u>set</u> x.Q .

No special notation will be introduced for specifications.

7. FORMAL DEVELOPMENT OF PROGRAMS

In this chapter we will show how programs can be formally derived using program descriptions. The use of program descriptions makes a formal proof of the correctness of the derivation 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 will 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 done can be proved. These proof rules will be formulated in the succeeding sections. Thus section 7.2 gives proof rules for proving the correctness of procedure implementations. Section 7.3 will give examples of transformation rules by which assertions can be introduced into descriptions. Section 7.4 will again give transformation rules, by which abstractions can be removed from descriptions. Finally, section 7.5 gives an example of a transformation rule, by which the control structure of a program description can be changed.

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.

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 of section 6.2.

The problem specification can be expressed as the program description

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

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

$$R_1: \quad X > 1 \wedge Y \geq 0$$

for future convienience. Thus A_0 is

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

We will assume that the variables take only integers as values. This means that we postulate some set Δ of sentences, which are taken as axioms and which give the operations used in the program descriptions the properties expected of the usual integer operations. In the sequel, this set Δ of axioms will not be mentioned explicitly. However, $S \leq S'$ in the sequel is to be understood as stating that $S \leq S'$ is a logical consequence of Δ, i.e. that $S \leq S'$ holds in any model of Δ.

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

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

The fact that $A_0 \leq A_1$ holds is a consequence of 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

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

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:

```
B1:  {R1};
     beg h:
          h,z:= X^Y,1; {R2} ;
          do  h ≠ 1   → set<h,z>. (h' < h ∧ R2');
                        {R2}
          od
     end
```

We have here used the abbreviations

$$R_2: \quad h \cdot z = X^Y \wedge h \geq 1 \qquad \text{and}$$

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

The assignment

$$\underline{\text{set}}\langle h,z\rangle .\ (h' < h \wedge R_2')$$

has the effect described in section 6.1 as

"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[76]. They come 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 the example of 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 neccessary 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};
                      set<h,z>.(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.

In the following, we will take a small shortcut in removing the representational abstraction h, as compared to the method outlined in in section 6.2. Thus we will not go through the recursive determination of the subproblems to be solved, but assume that this step is already done, leaving us with a number of sub-descriptions to be refined using abstractions. After giving these refinements, we use the transformation rules of section 7.4 to push the abstraction outwards until we can eliminate it completely.

We will consider the following two components of B_2:

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

D_0: $\{R_2 \wedge h \neq 1\}$; $\underline{set}\langle h,z\rangle.(h' < h \wedge R_2')$.

The program description C_0 will be implemented with the description

C_1: $\underline{beg}$ $\alpha\langle x,y\rangle\beta\langle h\rangle.Q$:

$x,y,z := X,Y,1$

$\underline{per}$,

where Q is

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

The effect of C_1 is to initialise 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 $\underline{beg}$ ... $\underline{per}$ is used here, because the initial value of h is not needed to compute the final 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: $\underline{rep}$ $\alpha\langle x,y\rangle\beta\langle h\rangle.Q$:

$y,z := y-1, z\cdot x$

$\underline{per}$.

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

Because $C_0 \leq C_1$ and $D_0 \leq D_1$, we are allowed to replace C_0 and D_0 in B_2 with C_1 and D_1, giving as a result the program description B_3, for which $B_2 \leq B_3$ holds. We have

```
B3:  {R1};
     beg h:
         beg α<x,y>β<h>.Q: x,y,z:= X,Y,1 per; {R2};
         do h ≠ 1 → rep α<x,y>β<h>.Q: y,z:= y-1,z·x per; {R2} od
     end.
```

We now apply transformation rules for abstractions, by which the operation of replacing the variable h with the variables x and y can be pushed outwards. We first use lemma 7.9, to push the abstraction out of the loop. This gives

```
B4:  {R1};
     beg h:
         beg α<x,y>β<h>.Q: x,y,z:= X,Y,1 per;
         rep α<x,y>β<h>.Q:
                do y ≠ 0 → y,z:=y-1,z·x od
         per
     end
```

Next we use lemma 7.7 for handling compound descriptions, giving

```
B5:  {R1};
     beg h:
         beg α<x,y>β<h>.Q:
                x,y,z:= X,Y,1;
                do y ≠ 0 → y,z:= y-1,z·x od
         per
     end
```

Finally we use lemma 7.10 to get rid of the now obsolete variable h which gives us

```
B6:  {R1};
     beg x,y:
          x,y,z:= X,Y,1;
          do y ≠ 0 → y,z:= y-1,z•x od
     end.
```

These transformation rules will be treated in section 7.4 below. As a result of the above transformations we have $B_3 \leq B_4 \leq B_5 \leq B_6$. Thus, all in all, we have found a refinement B_6 of B_0. The component B_0 of A_1 can therefore be replaced with B_6. This will then give us a solution to the programming problem posed. The solution A_2 will correspond to step S_2 in the example by Dijkstra.

```
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.
```

Moreover, we will have a formal proof of the correctness of A_2, i.e. of the fact that $A_0 \leq A_2$, when we give formal proofs of the correctness of the intermediate stages in the program development.

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

```
B3': {R1};
     beg h:
          {R1}; h,z:= X^Y,1;{R2};
          do h ≠ 1  → {R2 ∧ h ≠ 1}; skip;
                      {R2 ∧ h ≠ 1};
                      set<h,z>.(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, i.e. it does not do anything. 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 then also consider the component

$$E_0: \quad \{R_2 \wedge h \neq 1\};\ \text{skip}$$

of B_3'. This component will be implemented with the program description

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

The way in which $E_0 \leq E_1$ is proved is given in example 7.4 of section 7.2.

We then proceed as before, replacing in B_3' the components C_0, D_0 and E_0 with C_1, D_1 and E_1 respectively, and pushing the representation of h with x and y outwards, until it finally becomes possible to eliminate it altogether. As a result of 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' witch corresponds to the step S_3 by Dijkstra:

```
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 does make it simpler, by fusing the two nested loops of the program into one single loop. This transformation is not done by Dijkstra, for obvious reasons. We wish, however, to show here the usability of transformations of the control structure of programs, to make programs more efficient and/or easier to implement.

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 7.5, we get the refinement F_1 of F_0:

```
F1:   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 $y \neq 0$ is tested at each iteration, whereas 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

```
A3':  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 here give a general proof rule by which the correctness of an *implementation*, i.e. of a refinement of the form

$$\{P\};\ \underline{\text{set}}\ x.Q\ \leq\ S$$

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

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

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

when $var(Q) \subset V \cup \tilde{x}'$ and $var(R) \subset V$, $\tilde{x}_0$ 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 $\tilde{y} \subset V$.

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 $\{P\};\underline{set}\ x.Q$ and S be program descriptions 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 $$\Delta \vdash \{P\};\underline{set}\ x.Q \ \leq\ S.$$

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

$$\Delta \vdash WP(\{P\};\underline{set}\ x.Q,G(v)) \Rightarrow WP(S,G(v)).$$

Take therefore $WP(\{P\};\underline{set}\ x.Q,\ G(v))$ as an assumption, i.e. we assume that

$$P \wedge \exists x'.Q \wedge \forall x'(Q \Rightarrow G(v)[x'/x]). \qquad (7.3)$$

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

$$x=x_0 \wedge y=y_0. \qquad (7.4)$$

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(ii), 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 to 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 theorem once again, we eliminate assumption (7.3), getting the desired result

$$WP(\{P\};\underline{set}\ x.Q,\ G(v)) \Rightarrow WP(S,\ G(v)).\ \square$$

COROLLARY 7.3 Let the assumptions be as in the 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 $\vdash\ \underline{set}\ x.Q\ \underline{<}\ S.$

Proof: By noting that <u>set</u> x.Q $\approx$ {∃x'.Q};<u>set</u> 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

$$B_0: \{X > 1 \wedge Y \geq 0\};z:= X^Y.$$

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 initialisation, 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[76].

EXAMPLE 7.2 The second implementation was the implementation of C_0 with C_1, where

$$C_0: \{X > 1 \wedge Y \geq 0\}; h,z:= X^Y,1 \quad \text{and}$$

C_1: __beg__ α<x,y>β<h>.(h=x^y $\wedge$ x > 1): x,y,z:= X,Y,1 __per__.

This is again an initialising 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.. We have

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

Using it in the present example means that we have to prove that

$$X > 1 \wedge Y \geq 0 \Rightarrow \forall xy WP(x,y,z:= X,Y,1,(x > 1 \wedge x^y=X^Y \wedge z=1)).$$

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: {R_2 $\wedge$ h$\neq$1 }; __set__ <h,z>.(h' < h $\wedge$ R_2'),

and

D_1: __rep__ α<x,y>β<h>.(h=x^y $\wedge$ x > 1): y,z:= y-1,z$\cdot$x __per__

Here

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

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

We use the 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

$$R_2 \wedge h\neq 1 \wedge h=h_0 \wedge z=z_0 \Rightarrow WP(D_1, h < h_0 \wedge R_2)$$.

The weakest precondition for the abstraction D_1 is given by the formula

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

$$\Leftrightarrow \exists x(y=t \wedge I) \quad \wedge$$

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

Thus, in order to establish that $D_0 \leq D_1$, we have to prove that

$$R_2 \wedge h\neq 1 \wedge h=h_0 \wedge z=z_0$$

$$\Rightarrow \exists xy(h=x^y \wedge x > 1)$$

$$\wedge \quad \forall xy(h=x^y \wedge x > 1 \Rightarrow WP(y,z:=y-1,z\cdot x,(x > 1 \wedge x^y < h_0 \wedge R_2[x^y/h])))$$.

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

$$h\cdot z=X^Y \wedge h > 1 \wedge h=h_0 \wedge h=x^y \wedge x > 1$$

$$\Rightarrow \quad x > 1 \wedge x^{y-1} < h_0 \wedge x^{y-1}\cdot z\cdot x=X^Y \wedge x^{y-1} \geq 1.$$

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: $\{R_2 \wedge h\neq 1\}$;skip

and

E_1: <u>rep</u> $\alpha\langle x,y\rangle\beta\langle h\rangle.(h=x^y \wedge x > 1)$:
<u>do</u> $2|y \rightarrow x,y:=x\cdot x,y/2$ <u>od</u>
<u>per</u>.

To prove that $E_0 \leq E_1$, we can still use the theorem 7.2, because

$$\text{skip} \approx \underline{\text{set}}\ \langle\rangle.\text{true},$$

a fact that is easily verified. 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

$$E_1'\text{:}\ \ \underline{\text{do}}\ 2|y \rightarrow x,y{:=}\ x{\cdot}x,y/2\ \underline{\text{od}},$$

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 $\sim 2|y$ 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 formalised in this thesis. 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 developement, but will restrict ourselves to only giving examples of proof rules concerning assertions. 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

$$\Delta \vdash \{P\};\underline{do}\ B_1 \rightarrow S_1 | \ldots | B_n \rightarrow S_n\ \underline{od}^n \leq S, \quad \text{for } n < \omega,$$

then

$$\Delta \vdash \{P\};\underline{do}\ B_1 \rightarrow S_1 | \ldots | B_n \rightarrow S_n\ \underline{od} \leq S.$$

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

$$DO^n = \underline{do}\ B_1 \rightarrow S_1 | \ldots | B_n \rightarrow S_n\ \underline{od}^n \quad \text{and}$$

$$DO = \underline{do}\ B_1 \rightarrow S_1 | \ldots | B_n \rightarrow S_n\ \underline{od},$$

we have to prove that

$$WP(\{P\};DO^n, G(v)) \Rightarrow WP(S, G(v)) \quad \text{for } n < \omega \tag{7.4}$$

implies

$$WP(\{P\};DO, G(v)) \Rightarrow WP(S, G(v)). \tag{7.5}$$

The assumption (7.4) gives us that

$$P \wedge WP(DO^n, G(v)) \Rightarrow WP(S, G(v)), \text{ for } n < \omega,$$

or equivalently

$$P \Rightarrow (WP(DO^n, G(v)) \Rightarrow WP(S, G(v))), \text{ for } n < \omega.$$

Make the assumption P. We then have that

$$WP(DO^n, G(v)) \Rightarrow WP(S, G(v)), \text{ for } n < \omega ,$$

and using the inference rule for infinite disjunction, we get that

$$\bigvee_{n<\omega} WP(DO^n, G(v)) \Rightarrow WP(S, G(v)), \text{ i.e.}$$
$$WP(DO, G(v)) \Rightarrow WP(S, G(v)).$$

Using the deduction theorem, we get from this that

$$P \Rightarrow (WP(DO, G(v)) \Rightarrow WP(S,G(v))), \text{ i.e.}$$

$$P \wedge WP(DO, G(v)) \Rightarrow WP(S,G(v)),$$

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

$$WP(\{P\},G(v)) \Rightarrow WP(\{P'\},G(v)), \text{ i.e.}$$

$$P \wedge G(v) \Rightarrow P' \wedge G(v).$$

Because $\Delta \vdash P \Rightarrow \text{true}$ for any P, we have $\Delta \vdash \{P\} \leq \text{skip}$ 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

$$WP(\{P\};S,G(v)) \Leftrightarrow P \wedge WP(S,G(v))$$

and

$$P \wedge WP(S,G(v)) \Rightarrow P \wedge WP(S,Q) \wedge WP(S,G(v))$$

$$\Rightarrow P \wedge WP(S, Q \wedge G(v)) \quad \text{(lemma 5.11 (ii))}$$

$$\Leftrightarrow WP(\{P\};S;\{Q\}, G(v)).$$

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) $$\Delta \vdash \{P\};\underline{if}\ B_1 \rightarrow S_1 \mid \ldots \mid B_n \rightarrow S_n\ \underline{fi}$$
$$\approx \{P\};\underline{if}\ B_1 \rightarrow \{P \wedge B_1\};S_1 \mid \ldots \mid \{P \wedge B_n\};S_n\ \underline{fi}$$

and

(ii) $$\Delta \vdash \underline{if}\ B_1 \rightarrow S_1 \mid \ldots \mid B_n \rightarrow S_n\ \underline{fi};\{Q\}$$
$$\approx \underline{if}\ B_1 \rightarrow S_1;\{Q\} \mid \ldots \mid B_n \rightarrow S_n;\{Q\}\ \underline{fi}$$

also follow directly, by analysing the corresponding weakest preconditions.

EXAMPLE 7.8 We use the previous examples and lemma 7.5 to show that

$$\Delta \vdash \{P\};\underline{do}\ B_1 \rightarrow S_1;\{P\} \mid \ldots \mid B_n \rightarrow S_n;\{P\}\ \underline{od}$$
$$\approx \{P\};\underline{do}\ B_1 \rightarrow \{B_1 \wedge P\};S_1;\{P\} \mid \ldots \mid B_n \rightarrow \{B_n \wedge P\};S_n;\{P\}\ \underline{od}.$$

Denote the left hand side by $\{P\};DO$ and the right hand side by $\{P\};DO'$. By example 7.5, we only need to show that $\{P\};DO \leq \{P\};DO'$. Using the lemma 7.5, to show this, it is sufficient to show that

$$\{P\};DO^n \leq \{P\};DO', \text{ for } n < \omega.$$

Because $DO'^n \leq DO'$ for every $n < \omega$, it will be sufficient to show that

$$\{P\};DO^n \leq \{P\};DO'^n \quad \text{for } n < \omega. \tag{7.6}$$

For $n = 0$ this result is obvious, as $\{P\};DO^0 \approx$ abort. Assume that (7.6) holds for n, $n < \omega$.

By the definition of DO^n, we have that

$$\{P\};\ DO^{n+1}$$

$$= \{P\};\underline{if}\ BB \rightarrow \underline{if}\ B_1 \rightarrow S_1;\{P\} \mid \ldots \mid B_n \rightarrow S_n;\{P\}\underline{fi};DO^n$$
$$\mid \sim BB \rightarrow \text{skip}\ \underline{fi}$$

$$\leq \{P\};\underline{if}\ BB \rightarrow \{P\};\underline{if}\ B_1 \rightarrow S_1;\{P\} \mid \ldots \mid B_n \rightarrow S_n;\{P\}\ \underline{fi};DO^n$$
$$\mid \sim BB \rightarrow \text{skip}\ \underline{fi}$$

$$\leq \{P\};\underline{if}\ BB \rightarrow \underline{if}\ B_1 \rightarrow \{P \wedge B_1\};S_1;\{P\} \ldots$$
$$\mid B_n \rightarrow \{P \wedge B_n\};S_n;\{P\}\ \underline{fi};\{P\};DO^n$$
$$\mid \sim BB \rightarrow \text{skip}\ \underline{fi}$$

$$\leq \{P\};DO'^{n+1}.$$

In the first refinement above, we used example 7.7(i) and 7.5 (the latter e.g. when replacing $P \wedge BB$ with P, because $P \wedge B \Rightarrow P$). 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 assertion rules concerned with abstractions. The soundness of these rules can all be checked by considering the corresponding weakest preconditions, as was done in the preceeding examples. The rules are as follows, with $D = \alpha x\beta y.(y{=}t \wedge I)$:

(i) $\Delta \vdash \{P\};\underline{rep}\ D\colon S\ \underline{per};\{Q\}$

$\approx \{P\};\underline{rep}\ D\colon \{P[t/y] \wedge I\};\ S\ ;\{\ Q[t/y] \wedge I\}\ \underline{per};\{Q\}$,

(ii) $\Delta \vdash \{P\};\underline{rep}\ D\colon S\ \underline{end};\{Q\}$

$\approx \{P\};\underline{rep}\ D\colon \{P[t/y] \wedge I\};\ S\ ;\{\forall y Q\}\ \underline{end};\{Q\}$,

(iii) $\Delta \vdash \{P\};\underline{beg}\ D\colon S\ \underline{per};\{Q\}$

$\approx \{P\};\underline{beg}\ D\colon\{\exists y P\};\ S\ ;\ \{Q[t/y] \wedge I\}\ \underline{per};\{Q\}$, and

(iv) $\Delta \vdash \{P\};\underline{beg}\ \alpha x\beta y\text{: } S\ \underline{end};\{Q\}$

$$\approx \{P\};\underline{beg}\ \alpha x\beta y\text{:}\{\exists y P\};\ S\ ;\{\forall y Q\}\ \underline{end};\{Q\}.$$

The last case simplifies for the block to

(v) $\Delta \vdash \{P\};\underline{beg}\ x\text{: } S\ \underline{end};\{Q\}$

$$\approx \{P\};\underline{beg}\ x\text{:}\{P\};\ S\ ;\{Q\}\ \underline{end};\{Q\}.$$

because x cannot occur free in P or Q, and no variables are deleted, i.e. y = <>.

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 rule (v) in example 7.9 and the rule in example 7.8.

7.4 Transformation rules for abstractions

In this subchapter we give some rules for handling abstractions in program descriptions. The purpose of these rules is to enable one to eliminate an abstraction that has been previously introduced into a program description. As in the previous subchapters 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 not always be in the most general form possible.

For the formulation of the results below, we will fix below a countable set Δ of sentences of L. We also have a finite nonempty set V of program variables. Let $D = \alpha x \beta y.(y=t \wedge I):V \rightarrow W$ be a description. Here $var(t) \subset W$ and $var(I) \subset W$. Also, let

$$S_i = \underline{rep}\ D:\ S_i'\ \underline{per},\ \text{for } i = 1,\dots,n$$

where $n \geq 1$.

LEMMA 7.6 If $\Delta \vdash P \Rightarrow \exists x(y=t \wedge I)$, then

$$\Delta \vdash \{P\};skip \leq \underline{rep}\ D:\ skip\ \underline{per}.$$

Proof: The case here is similar to the case in example 7.4, and we have to prove that

$$P \wedge y=y_0 \Rightarrow WP(\underline{rep}\ D:\ skip\ \underline{per},\ y=y_0). \qquad (7.7)$$

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 already 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 (7.7).

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 $\Delta \vdash S_1;\ldots;S_n \leq \underline{\text{rep}}\ D\colon S_1';\ldots;S_n'\ \underline{\text{per}}$, for $n \geq 2$.

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

$$WP(S_1;S_2,\ G(v)) \Rightarrow WP(\underline{\text{rep}}\ D\colon S_1';S_2'\ \underline{\text{per}},\ G(v)).$$

First, we have that

$$\begin{aligned} WP(S_2,\ G(v)) &\Leftrightarrow \exists x(y=t \wedge I) \wedge \forall x(y=t \wedge I \Rightarrow WP(S_2', I \wedge G(v)[t/y])) \\ &\Leftrightarrow P_1 \wedge P_2, \end{aligned}$$

and

$$\begin{aligned} WP(S_1,\ P_1 \wedge P_2) \Leftrightarrow\ & \exists x(y=t \wedge I) \\ & \wedge\ \forall x(y=t \wedge I \Rightarrow WP(S_1',\ I \wedge P_1[t/y] \wedge P_2[t/y])). \end{aligned}$$

We concentrate on the formula $P_2[t/y]$. Changing the bound variable x to a fresh variable x' gives

$$P_2 \Leftrightarrow \forall x'(y=t[x'/x] \wedge I[x'/x] \Rightarrow WP(S_2', I \wedge G(v)[t/y])[x'/x]),$$

thus making t free for y in the formula P_2. Thus we have that

$$P_2[t/y] \Leftrightarrow \forall x'(t=t[x'/x] \wedge I[x'/x] \Rightarrow WP(S_2', I \wedge G(v)[t/y])[x'/x])$$

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 generalisation 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(ii), and get

$$\begin{aligned} & WP(S_1', I \wedge P_1[t/y] \wedge P_2[t/y]) \\ & \qquad \Rightarrow WP(S_1', WP(S_2', I \wedge G(v)[t/y])) \\ & \qquad \Leftrightarrow WP(S_1';S_2', I \wedge G(v)[t/y]). \end{aligned}$$

Thus we have that

$$\begin{aligned} & WP(S_1;S_2, G(v)) \Leftrightarrow WP(S_1, P_1 \wedge P_2) \\ & \qquad \Leftrightarrow \exists x(y=t \wedge I) \wedge \forall x(y=t \wedge I \Rightarrow WP(S_1', I \wedge P_1[t/y] \wedge P_2[t/y])) \\ & \qquad \Rightarrow \exists x(y=t \wedge I) \wedge \forall x(y=t \wedge I \Rightarrow WP(S_1';S_2', I \wedge G(v)[t/y])) \\ & \qquad \Leftrightarrow WP(\underline{rep}\ D: S_1';S_2', G(v)), \end{aligned}$$

as required. □

LEMMA 7.8 Let $S = \underline{\text{if}}\ B_1 \rightarrow S_1\ |\ldots|\ B_n \rightarrow S_n\ \underline{\text{fi}}$ and

$S' = \underline{\text{if}}\ B_1' \rightarrow S_1'\ |\ldots|\ B_n' \rightarrow S_n'\ \underline{\text{fi}}$.

Assume that $\Delta \vdash P \Rightarrow \exists x(y=t \wedge I)$, and further that

$$\Delta \vdash P \wedge y=t \wedge I \Rightarrow (B_i \Leftrightarrow B_i'),\ \text{for } i = 1,\ldots,n.$$

Then $\Delta \vdash \{P\};S \leq \underline{\text{rep}}\ D\colon S'\ \underline{\text{per}}$. Here S_i, S_i' and D are assumed to be as stated at the beginning of the this subchapter.

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

$$P \wedge BB \wedge \bigwedge_{1\leq i\leq n} (B_i \Rightarrow WP(S_i, G(v))). \tag{7.8}$$

We have to prove that $WP(\underline{\text{rep}}\ D\colon S'\ \underline{\text{per}},\ G(v))$ 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 (7.8) because of the assumption, so we only need to prove that the second conjunct also is implied. Assume therefore that

$$y=t \wedge I. \tag{7.9}$$

Then $B_i \Rightarrow B_i'$ by the assumption, for $i = 1,\ldots,n$, so we get from (7.8) 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 (7.8), 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 (7.9) gives

$$y=t \wedge I \Rightarrow WP(S', I \wedge G(v)[t/y]),$$

and as x is not free in assumption (7.8), we have

$$\forall x(y=t \wedge I \Rightarrow WP(S', I \wedge G(v)[t/y])),$$

thus concluding the proof. □

LEMMA 7.9 Let $DO = \underline{do}\ B_1 \rightarrow S_1;\{P\} | \ldots | B_n \rightarrow S_n;\{P\}\ \underline{od}$, $P' = P[t/y] \wedge I$
and $DO' = \underline{do}\ B_1' \rightarrow S_1';\{P'\} | \ldots | B_n' \rightarrow S_n';\{P'\}\ \underline{od}$.
Assume that $\Delta \vdash P \Rightarrow \exists x(y=t \wedge I)$, and further that

$$\Delta \vdash P \wedge y=t \wedge I \Rightarrow (B_i \Leftrightarrow B_i'), \text{ for } i = 1,\ldots,n.$$

Then $\Delta \vdash \{P\};DO \leq \underline{rep}\ D: DO'\ \underline{per}$. Here S_i, S_i' and D are assumed to be as stated at the beginning of this subchapter.

Proof: Let DO^n and DO'^n have their usual meaning. We will prove that

$$\{P\};DO^n \leq \underline{rep}\ D: DO'^n\ \underline{per}\ , \text{ for } n < \omega\ . \qquad (7.10)$$

Because $DO'^n \leq DO'$, this will give us that

$$\{P\};DO^n \leq \underline{rep}\ D: DO'\ \underline{per}\ , \text{ for } n < \omega,$$

from which the desired result then follows using lemma 7.5.

For n = 0, (7.10) obviously holds, as DO^0 = abort. Assume that (7.10) holds for n, n ≥ 0. We have that

$$\{P\};DO^{n+1} = \{P\};\underline{if}\ BB \rightarrow \underline{if}\ B_1 \rightarrow S_1;\{P\} \mid \ldots \mid B_n \rightarrow S_n;\{P\}\ \underline{fi};\ DO^n$$
$$\mid \sim BB \rightarrow skip\ \underline{fi}$$
$$\leq \{P\};\underline{if}\ BB \rightarrow \{P\};\underline{if}\ B_1 \rightarrow S_1;\{P\} \mid \ldots \mid B_n \rightarrow S_n;\{P\}\ \underline{fi};\{P\};DO^n$$
$$\mid \sim BB \rightarrow \{P\};skip\ \underline{fi}\ ,$$

Using the rules for assertions discussed in subchapter 7.2.

By lemma 7.6, we have

$$\{P\};skip \leq \underline{rep}\ D{:}\ skip\ \underline{per}, \tag{7.11}$$

and using example 7.9(i) we have that

$$\{P\};skip \leq \underline{rep}\ D{:}\{P'\};skip\ \underline{per}.$$

This means that

$$\{P\} \leq \underline{rep}\ D{:}\ \{P'\}\ \underline{per},$$

because {P};skip ≈ {P} for any P.

Now, using lemma 7.7 we get that

$$S_i;\{P\} \leq \underline{rep}\ D{:}\ S_i';\{P'\}\ \underline{per},\ \text{for } i = 1,\ldots,n.$$

And using lemma 7.8, we get from this that

$$\{P\};\underline{if}\ B_1 \rightarrow S_1;\{P\} \mid \ldots \mid B_n \rightarrow S_n;\{P\}\ \underline{fi}$$
$$\leq \underline{rep}\ D{:}\ \underline{if}\ B_1' \rightarrow S_1';\{P'\} \mid \ldots \mid B_n' \rightarrow S_n';\{P'\}\ \underline{fi}\ \underline{per}$$

Finally, using induction hypothesis (7.10), result (7.11), lemma 7.7 and lemma 7.8 again, we finally get the result

$$\{P\};DO^{n+1} \leq \underline{rep}\ D{:}\ DO'^{n+1}\ \underline{per},$$

thus proving that (7.10) holds for each n < ω. □

The transition step leading from B_3 to B_4 in the example of section 7.1 can be justified by the lemma 7.9. In order to apply the required transformation we need to prove the following two conditions:

$$\text{(i)}\quad R_2 \Rightarrow \exists x,y.\ (h = x^y \wedge x > 1)\quad \text{, and}$$

$$\text{(ii)}\quad R_2 \wedge h = x^y \wedge x > 1 \Rightarrow (h \neq 1 \Leftrightarrow y \neq 0)$$

Writing R_2 explicitly, we get the formulas

$$\text{(i)}\quad h \cdot z = X^Y \wedge h \geq 1 \Rightarrow \exists x,y.(h = x^y \wedge x > 1)\quad \text{, and}$$

$$\text{(ii)}\quad h \cdot z = X^Y \wedge h \geq 1 \wedge h = x^y \wedge x > 1 \Rightarrow (h \neq 1 \Leftrightarrow y \neq 0)\ .$$

These formulas are readily seen to be true.

The transition from B_4 to B_5 is justified by lemma 7.7, by noting that this lemma still holds when S_1 = beg D: S_1' per (this is readily seen by inspecting the proof of the lemma). To prove the final step from B_5 to B_6, where the abstraction is eliminated, we need the following lemma.

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 $\tilde{y} \subset \tilde{y}'$. Then

$$\Delta \vdash \underline{\text{beg}}\ y'\text{: } \underline{\text{beg}}\ D\text{: } S\ \underline{\text{per}}\ \underline{\text{end}} \leq \underline{\text{beg}}\ y'',x\text{: } S\ \underline{\text{end}},$$

where $\tilde{y}'' = \tilde{y}' - \tilde{y}$, $S:W \rightarrow W$ and $D = \alpha x \beta y.(y = t \wedge I):V \rightarrow W$ 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

$$\forall y' WP(\underline{\text{beg}}\ D\text{: } S\ \underline{\text{per}}, G(v')) \ ,$$

which again is equivalent to

$$\forall y' x WP(S,\ I \wedge G(v')[t/y]).$$

Now, because $\tilde{y} \cap V' = \emptyset$, as $\tilde{y} \subset \tilde{y}'$, 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(ii), this gives us that

$$WP(S,\ I \wedge G(v')[t/y]\ ) \Rightarrow WP(S,\ G(v')),$$

i.e. the assumption gives us that

$$\forall y' x WP(S,G(v')).$$

As y cannot occur free in WP(S, G(v')), because $S\text{:}W \rightarrow W$, and $\tilde{y} \cap W = \emptyset$, this is again equivalent to

$$\forall y'' x WP(S,G(v')),$$

which, from the definition of WP, is

$$WP(\underline{\text{beg}}\ y'',x\text{: } S\ \underline{\text{end}},\ G(v')).$$

This proves the lemma. □

7.5 Transformation rules for control structures

We finally 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

$$\mathrm{DO} = \underline{\mathrm{do}}\ B \rightarrow \mathrm{DO}';S\ \underline{\mathrm{od}},$$

$$\mathrm{DO}' = \underline{\mathrm{do}}\ B' \rightarrow S'\ \underline{\mathrm{od}} \quad \text{and}$$

$$\mathrm{DO}'' = \underline{\mathrm{do}}\ B \rightarrow \underline{\mathrm{if}}\ B' \rightarrow S' \mid \sim B' \rightarrow S\ \underline{\mathrm{fi}}\ \underline{\mathrm{od}}\ .$$

Assume that $\{B \wedge B'\};S' \leq \{B \wedge B'\};S';\{B\}$. Then $\mathrm{DO} \leq \mathrm{DO}''$.

We first show that

$$\{B\};\mathrm{DO}'^{n};S;\mathrm{DO} \leq \{B\};\mathrm{DO}'', \quad \text{for } n < \omega. \qquad (7.12)$$

For $n = 0$ this is obvious, as $\{B\};\mathrm{DO}'^{0};S;\mathrm{DO} \approx$ abort. Assume that (7.12) holds for n, $n < \omega$. We have that

$$\{B\};\mathrm{DO}'^{n+1};S;\mathrm{DO} = \{B\};\underline{\mathrm{if}}\ B' \rightarrow S';\mathrm{DO}'^{n} \mid\ \sim B' \rightarrow \mathrm{skip}\ \underline{\mathrm{fi}};S;\mathrm{DO}.$$

Consider now separately the two cases $B \wedge B'$ and $B \wedge \sim B'$.

(i) $B \wedge B'$. We have that

$$\{B \wedge B'\};\mathrm{DO}'^{n+1};S;\mathrm{DO} \leq \{B \wedge B'\};S';\mathrm{DO}'^{n};S;\mathrm{DO}$$

$$\leq \{B \wedge B'\};S';\{B\};\mathrm{DO}'^{n};S;\mathrm{DO}$$

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

$$\begin{aligned}\{B \wedge B'\};S';DO'' &\leq \{B \wedge B'\};\underline{if}\ B \to \underline{if}\ B' \to S';DO'' \\ &\qquad\qquad\qquad\qquad \mid \sim B' \to S\ ;DO''\ \underline{fi} \\ &\qquad\qquad\quad \mid \sim B \to skip\ \underline{fi} \\ &\leq \{B \wedge B'\};\underline{if}\ B \to \underline{if}\ B' \to S' \mid \sim B' \to S\ \underline{fi}; \\ &\qquad\qquad\qquad\qquad DO'' \\ &\qquad\qquad\quad \mid \sim B \to skip\ \underline{fi} \\ &\leq \{B \wedge B'\};DO''.\end{aligned}$$

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

$$\begin{aligned}\{B \wedge \sim B'\};S;DO &\leq \{B \wedge \sim B'\};\{B\};DO'^{n};S;DO \\ &\leq \{B \wedge \sim B'\};DO'',\end{aligned}$$

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

$$\{B\};DO';S;DO \leq \{B\};DO''. \qquad (7.13)$$

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

$$DO^n \leq DO'', \text{ for } n < \omega. \tag{7.14}$$

For n = 0 this is immediate, as usual. Assume that (7.14) holds for n, n < ω. We then have that

$$\begin{aligned} DO^{n+1} &= \underline{if}\ B \to DO';S;DO^n \mid \sim B \to skip\ \underline{fi} \\ &\leq \underline{if}\ B \to \{B\};DO';S;DO \mid \sim B \to skip\ \underline{fi} \\ &\leq \underline{if}\ B \to \{B\};DO'' \mid \sim B \to skip\ \underline{fi} \\ &\leq \underline{if}\ B \to \underline{if}\ B' \to S' \mid \sim B' \to S\ \underline{fi};DO'' \\ &\qquad \mid \sim B \to skip\ \underline{fi} \\ &\leq DO''. \end{aligned}$$

In these steps we have made use of the fact that

$$\begin{aligned} DO'' \approx\ & \underline{if}\ B \to \underline{if}\ B' \to S' \mid \sim B' \to S\ \underline{fi};DO'' \\ & \mid \sim B \to skip\ \underline{fi}. \end{aligned}$$

The derivation shows that (7.14) holds, thus proving the desired result.

8. REFERENCES

Burstall & Darlington[75]. Some transformations for developing recursive programs. Sigplan Notices 10, 6, 1975.

Correll[78]. Proving programs correct trough refinement. Acta Informatica 9, 1978.

de Bakker[77a]. Topics in denotational semantics. Lecture Notes for the Advanced Summer School on Mathematical Foundations of Computer Science, Turku 1977.

de Bakker[77b]. Recursive programs as predicate transformers. Matematisch Centrum IW 83/77, June 1977.

Dijkstra[68]. A constructive approach to the problem of program correctness. BIT 8, 1968.

Dijkstra[72]. Notes on structured programming. In: Structured Programming, Academic Press, New York, 1971.

Dijkstra[76]. A Discipline of Programming. Prentice-Hall, 1976.

Engeler[75].Algorithmic logic. Found. Comp. Sc., Mathematical Centre Tracts 63.

Feferman[68]. Lectures in proof theory. In: Proceedings of the Summer School in Mathematical Logic, Leeds 1967. Springer Verlag 1968.

Gerhardt[75]. Correctness preserving program transformations. Second ACM Conference on Principles of Programming Languages, 1975.

Harel & al[77]. A complete axiomatic system for proving deductions about recursive programs. Proc. 9th annual ACM Symp. on the Theory of Computing, Boulder, Colorado. May 1977.

Hoare[69]. An axiomatic basis for computer programming. Comm. ACM 12, 10, October 1969.

Hoare[71]. Procedures and parameters: An axiomatic approach. Symposium on Semantics of Algorithmic Languages. Springer Verlag 1971.

Hoare[72]. Proof of correctness of data representation. Acta Informatica 1, 1972.

Hoare & Wirth[73]. An axiomatic definition of the programming language Pascal. Acta Informatica 2, 1973.

Karp[64]. Languages with Expressions of Infinite Length. North-Holland, 1964.

Katz & Manna[76]. Logical analysis of programs. Comm. ACM 19, 4, 1976.

Keisler[71]. Model Theory for Infinitary Logic. North-Holland, 1971.

Knuth[74]. Structured programming with the go to statement. Computing Surveys 6, 4, 1974.

Lampson & al[77]. Report on the programming language Euclid. Sigplan Notices 12, 2, 1977.

Liskov & al[77]. Abstraction mechanism in Clu. Comm. ACM 20, 8, August 1977.

Loveman[77]. Program improvement by source to source transformations. J. ACM 24, 1, January 1977.

Manna[74]. Mathematical Theory of Computing. McGraw-Hill, 1974.

Milner[71]. An algebraic definition of simulation between programs, CS 205, Stanford University, February 1971.

Plotkin[76]. A powerdomain construction. SIAM J. of Computing 5, 3, September 1976.

Scott[65]. Logic with denumerably long formulas and finite strings of quantifiers. Symp. on the Theory of Models, North-Holland, 1965.

Smythe[76]. Powerdomains. Mathematical Foundations of Computer Science, Springer Verlag, 1976.

Wegbreit[76]. Goal directed program transformations. IEEE transactions on Software Engineering, SE-2, 2, 1976.

Wirth[71]. Program development by stepwise refinement. Comm. ACM 14, 4, May 1971.

Wirth[73]. Systematic Programming. Prentice-Hall, 1973.

Wirth[77]. Modula, a language for modular multiprogramming. Software-Practice and Experience 7, 1, 1977.

Wulf & al[77]. An introduction to the construction and verification of Alphard programs. IEEE Transactions on Software Engineering SE-2, 4, 1977.