Acta Informatica 25, 593–624 (1988)

Acta Informatica

A Calculus of Refinements for Program Derivations

R.J.R. Back
Abo Akademi, Department of Computer Science, Lemminkäisenk. 4, SF-20520, Turku, Finland

Summary. A calculus of program refinements is described, to be used as a tool for the step-by-step derivation of correct programs. A derivation step is considered correct if the new program preserves the total correctness of the old program. This requirement is expressed as a relation of (correct) refinement between nondeterministic program statements. The properties of this relation are studied in detail. The usual sequential statement constructors are shown to be monotone with respect to this relation and it is shown how refinement between statements can be reduced to a proof of total correctness of the refining statement. A special emphasis is put on the correctness of replacement steps, where some component of a program is replaced by another component. A method by which assertions can be added to statements to justify replacements in specific contexts is developed. The paper extends the weakest precondition technique of Dijkstra to proving correctness of larger program derivation steps, thus providing a unified framework for the axiomatic, the stepwise refinement and the transformational approach to program construction and verification.

1. Introduction

Research in programming methodology has been very fruitful in the last twenty years. Early work on program verification by Floyd [Fl67], Naur [Na66] and Hoare [Ho69] established a mathematically sound basis for studying program construction methods. Stepwise refinement was proposed as a systematic program derivation method by Dijkstra and Wirth in the early seventies [Di71, Wi71]. Other paradigms were the program transformation approach, which grew out of this, by Gerhart [Ge75] and Burstall and Darlington [BuDa77]. A third approach, based on identifying the invariant and termination function of loops, was inspired by Hoare's axiomatization of partial correctness [Ho71] and Dijkstra's work on the weakest precondition calculus [Di76] and has been further studied and developed by Gries [Gr81], Reynolds [Re81] and Hehner [He84], among others.

The *refinement calculus* is an attempt to unify the stepwise refinement and program transformation approaches with the invariant-based approach to program construction. The basic notion in this calculus is the relation of (*correct*)

refinement between programs. A program S is said to be correctly refined by another program S' if S' *preserves the correctness of* S, in the sense that S' satisfies any specification that S satisfies.

Specifications are treated as program statements in the refinement calculus. They may not necessarily be executable, but they determine a well-defined effect on the program state. A specification need not determine the required effect uniquely, it may only state some properties that the result must satisfy. Hence specifications are nondeterministic statements.

Treating specifications as program statements is a conceptual simplification, since there is then only one kind of entity to work with. It does introduce some complications in the semantics and proof theory, since certain usual assumptions about statements are no longer necessarily valid. The assumption of bounded nondeterminacy is probably the most important one. This says that any nondeterministic program that can have an infinite number of possible final states must also be potentially nonterminating. This property, initially identified in [Di76], is an inherent property of executable nondeterministic program statements. Specifications need not, however, have this property, so programs of unbounded nondeterminism are permitted in the refinement calculus.

The refinement calculus is thus concerned with the relationship of correct refinement between nondeterministic program statements. The notion of correct refinement is applicable to any kind of program derivation step in which a new version of a program is produced that must be logically consistent with the previous version. A derivation step can for instance consist of implementing an input-output specification by a program, applying a program transformation rule, removing recursion from a program or changing the data representation in the program. A derivation step can be applied to the whole program or to only some part of it. The refinement calculus provides a framework in which the correctness of specific derivation steps can be established and in which general methods and rules of inference for program derivations can be proved correct.

The refinement calculus was originally described in [Ba78, Ba80, Ba81b], where a simple base language for programs and specifications was proposed and the relation of correct refinement was defined and studied. Our aim in this paper is to study the relation of correct refinement between nondeterministic statements further and in more detail. We will not be concerned with the different methods by which derivation steps can be carried out and their formalization in the refinement calculus, as this topic is covered quite extensively in [Ba80], but concentrate on developing the mathematical basis for this calculus.

The notion of correct refinement can be defined generally for any class of programs and specifications as follows [Ba81b]. Let Stat be a set of program statements and Spec the set of specifications to which these statements can be compared. Let sat be the satisfaction relation: if S is in Stat and R in Spec, then S sat R means that statement S satisfies specification R.

Let S be some statement and S' another statement produced by a derivation step from S. The derivation step preserves correctness if S' satisfies any specification that S satisfies:

$$S \text{ sat } R \Rightarrow S' \text{ sat } R, \quad \text{for any } R \in \text{Spec}.$$

We say that S is (*correctly*) *refined* by S', denoted by S ref S', if this condition holds.

It follows immediately from the definition that ref is reflexive and transitive, i.e. it is a preorder. Hence, successive refinements can be carried out on some initial program statement: if S_0 is the original program statement and S_1, S_2, ..., S_n are successive refinements, then

$$S_0 \text{ ref } S_1 \text{ ref} \ldots \text{ref } S_n \quad \text{implies } S_0 \text{ ref } S_n.$$

To be useful in program derivation, the constructs for combining statements into larger statements must be monotonic with respect to ref: if $S[T]$ is some program statement containing substatement T, then

$$T \text{ ref } T' \Rightarrow S[T] \text{ ref } S'[T'] \quad \text{for any statement } T'.$$

A substatement can thus always be replaced by its refinement. This means that program derivation can be carried out in a top-down manner: a derivation step is applied to some part of the program, and if this step is correct, then the original part may be replaced by the result of the derivation. Monotonicity depends on the class of program statements and program specifications considered and on the way in which the satisfaction relation is defined; it may or many not hold for a specific choice of these.

The refinement calculus considers the relation of correct refinement for a particular choice of program statements, nondeterministic statements with recursion, and uses total correctness as the criteria of correctness. With these choices, S ref S', if

$$P\langle S\rangle Q \Rightarrow P\langle S'\rangle Q$$

holds for any pre-postcondition pair (P, Q). Here $P\langle S\rangle Q$ states that statement S is totally correct with respect to precondition P and postcondition Q.

The contents of the paper is as follows. We present the necessary background on weakest preconditions and strongest postconditions in Sect. 2.

A simple programming language is introduced in Sect. 3. This language generalizes the guarded commands of Dijkstra [Di76] by adding recursion and permitting basic statements of unbounded non-determinism. Predicate transformers are defined for these constructs.

The notion of correct refinement is studied in Sect. 4. The relation is defined in such a way that it permits introducing auxiliary variables during program derivation. In Sect. 5 we prove that our statement are indeed monotone with respect to the refinement relation.

In Sect. 6 we show how refinement between statements can be proved in the weakest precondition calculus. We derive a proof rule that reduces the task of proving that a statement S is refined by another statement S' to proving that S' is totally correct with respect to a specific pre-postcondition pair, which depends on the statement S being refined. This also provides us with an alternative characterization of refinement.

In Sect. 7 we consider a subclass of refinements called *conservative refinements*. These are simpler to use, but do not permit the free introduction of auxiliary variables.

In Sect. 8 we study *context dependent* replacements. These are replacements that are correct in some contexts but not in all. Such replacements cannot therefore be justified by monotonicity alone. We show how the context of a replacement can be taken into account by introducing *context assertions* into statements.

The method of introducing context assertions is related to the techniques for proving partial and total correctness of programs. We study the relationship between these notions in Sect. 9 and relate the method to the use of proof outlines in program derivations.

The use of the refinement calculus is illustrated in Sect. 10 with the derivation of a small example program. The example is intended only to illustrate the main ideas of the paper, without being too big. A more substantial application is described in [Ba88], where the method is applied to the formal derivation of a marking algorithm originally constructed by informal derivations in [BaMaRa83].

The basic ideas of the refinement calculus were originally presented in [Ba78, Ba80]. In this paper, we develop further the basic foundations of this calculus. The method is improved in a number of ways, and most results are either new or substantial generalizations of the previous results. The language is extended to include full recursion, and unbounded nondeterminism is permitted without restriction. The connection with the predicate transformer approach is further emphasized, and strongest postconditions are introduced into the theory. The characterization and proof rule for refinement of Sect. 6 is new. It provides a simple bridge between the refinement and the weakest precondition calculi. The definition of refinement is extended to permit the introduction of auxiliary variables. The method for context-dependent replacements has been reworked completely, and its connection with partial and total correctness is studied in detail.

The general criteria for correctness preserving refinement is defined and studied also for correctness criteria other than total correctness in [Ba81b]. The connection between weakest preconditions and unbounded nondeterminism was previously studied in [Ba81a].

There has been relatively little work on general relations of refinement between programs. In most papers on program transformations, correctness is simply taken to be semantic equivalence of programs. This is the approach used in [BuDa77]. The approach closest to ours is the one used in the CIP project [BBPW79], where the program derivations are studied within a functional framework, based on the algebraic semantics. Nondeterministic functions are permitted, and the nondeterminism may be unbounded. However, the criteria for correctness of a refinement step is different. There is also no connection with the weakest precondition approach in their work. In [BrPeWi80] a number of possible relations between programs are defined, but the specific relation of refinement used here is not included. An approach that is close in spirit to ours is that by Hoare and Hehner on programs as specifications [He84b,

Ho85]. Their approach is in a sense dual to the one taken here: we treat specifications as programs, they treat programs as specifications. The resulting calculi are, however, very different. Our refinement relation turns out to be essentially the same as the Smyth order [Sm78], which was used to provide a denotational semantics for bounded nondeterminism. The purpose and applications of that work is very different from ours, although it is interesting that essentially the same basic notion works both in the semantics of nondeterminism and in the theory of correct program refinements (see [Pl79, Ba81b] for further details of this connection).

2. Predicate Transformers

We base our approach to correct refinement on predicate transformers. Since we are primarily interested in the total correctness of statements, weakest preconditions will be central in our study. However, strongest postconditions turn out to be useful also, because of their nature as forward predicate transformers. The weakest precondition approach is due to Dijkstra [Di76], and is also explained and further developed in [Gr81, He84, JaGr85].

2.1 Semantics of Statements and Predicates

Assume that an infinite set of (*program*) *variables* Var is given, each variable taking its value in a fixed set Val of values. Let υ be some set of program variables. A *state of* υ is an assignment of values to the variables in υ, i.e. a function $\sigma: \upsilon \to \mathrm{Val}$. We write Σ_υ for the set of all states of υ (the *state space of* υ).

A (*nondeterministic*) *state transformation* on υ assigns to each *initial state* σ of υ a nonempty set of possible *final states.* A final state is either a state of υ or the special *undefined state* $\perp$. A state transformation on υ is thus a function $f: \Sigma_\upsilon \to P(\Sigma_\upsilon \cup \{\perp\})$, where $f(\sigma)$ is nonempty for each initial state σ. A *state condition on* υ is a subset $p \subseteq \Sigma_\upsilon$.

The nondeterminism of a state transformation f is said to be *bounded* if for each initial state σ, either $f(\sigma)$ is finite or $\perp \in f(\sigma)$. Otherwise, the nondeterminism of f is *unbounded.*

Let S be a program statement and let $\mathrm{var}(S)$ denote the set of program variables of S. We write $S: \upsilon$ for the *restriction* of statement S to the *variable environment* υ, where $\mathrm{var}(S) \subseteq \upsilon$, i.e. S is considered as a statement on the variables υ only. The meaning of a (restricted) statement $S: \upsilon$ is a state transformation $f_{S,\upsilon}$ on υ, which describes the effect of executing S in an environment with program variables υ. For an initial assignment of values σ to υ, $f_{S,\upsilon}(\sigma)$ is the set of possible final assignments of values to these same program variables, when the execution of S terminates. In addition, $f_{S,\upsilon}(\sigma)$ contains the undefined state $\perp$ if and only if it is possible that execution does not terminate for this initial state.

Let R be a predicate (logical formula), and let $\mathrm{var}(R)$ denote the free variables of R. We write $R\colon \upsilon$ for the *restriction* of R to the variable environment υ, where $\mathrm{var}(S)\subseteq\upsilon$, i.e. R is considered as a condition on the variables υ only. The meaning of a (restricted) predicate $R\colon \upsilon$ is a state condition $p_{R,\upsilon}$ on υ. This is the set of all states of υ that satisfy R.

For notational simplicity, we sometimes identify a statement $S\colon \upsilon$ with the state transformation $f_{S,\upsilon}$ that it denotes and a predicate $R\colon \upsilon$ with the state condition $p_{S,\upsilon}$ that it denotes. A state σ of υ is identified with the list of values assigned to the variables in υ. Thus, we talk about the initial state υ_0 of υ, where υ_0 is a list of values, and about the set of final states $S(\upsilon_0)$ of S for this initial state.

2.2 *Weakest Preconditions and Strongest Postconditions*

Weakest Preconditions. Let S be a program statement and Q a predicate. The predicate $\mathrm{wp}(S, Q)$ is the *weakest precondition* that guarantees that execution of S will terminate in a final state that satisfies Q. We assume that $\mathrm{var}(\mathrm{wp}(S, Q))\subseteq\upsilon$ if $\mathrm{var}(S)\subseteq\upsilon$ and $\mathrm{var}(Q)\subseteq\upsilon$, for any set of program variables υ. Hence, if S and Q are restricted to variable environment υ, then $\mathrm{wp}(S, Q)$ may also be restricted to υ. The meaning of the weakest precondition is then given by

$$\mathrm{wp}(S, Q)(\upsilon_0) \Leftrightarrow S(\upsilon_0)\subseteq Q \qquad \text{for each state } \upsilon_0 \text{ of } \upsilon.$$

Note that Q does not contain the element $\perp$, so $S(\upsilon_0)$ does not contain $\perp$ either, i.e. termination of S for initial state υ_0 is guaranteed.

The following properties hold for the weakest precondition of the statement $S\colon \upsilon$, where $P, Q\colon \upsilon$ are arbitrary conditions [Di76]:

$$\mathrm{wp}(S, \mathrm{false}) \Leftrightarrow \mathrm{false} \tag{1}$$

$$\forall \upsilon \cdot (P \Rightarrow Q) \Rightarrow (\mathrm{wp}(S, P) \Rightarrow \mathrm{wp}(S, Q)) \tag{2}$$

$$\mathrm{wp}(S, P) \wedge \mathrm{wp}(S, Q) \Leftrightarrow \mathrm{wp}(S, P \wedge Q) \tag{3}$$

$$\mathrm{wp}(S, P) \vee \mathrm{wp}(S, Q) \Rightarrow \mathrm{wp}(S, P \vee Q) \tag{4}$$

The last two properties generalize to arbitrary disjunctions and conjunctions (even infinite). We do not assume the continuity property for weakest preconditions, since it is only valid if the nondeterminism is bounded.

Weakest preconditions are used to formalize the notation of *total correctness*: A statement $S\colon \upsilon$ is totally correct with respect to precondition $P\colon \upsilon$ and postcondition $Q\colon \upsilon$, denoted by $P\langle S\rangle Q$, if $P \Rightarrow \mathrm{wp}(S, Q)$.

Strongest Postcondition. Given a program statement S and a predicate P, the predicate $\mathrm{sp}(S, P)$ is the *strongest (liberal) postcondition* that holds of the final state of an execution of S when the execution terminates, if the execution is started in some initial state that satisfies P. We assume that $\mathrm{var}(\mathrm{sp}(S, P))\subseteq\upsilon$, if $\mathrm{var}(S)\subseteq\upsilon$ and $\mathrm{var}(P)\subseteq\upsilon$, for any set of program variables υ. Hence, if S and

P are restricted to the variable environment v, then $\mathrm{sp}(S, P)$ may also be restricted to it. The meaning of the strongest postcondition is then given by

$$\mathrm{sp}(S, P)(v_1) \Leftrightarrow \exists v_0 \in P \cdot v_1 \in S(v_0), \qquad \text{for each state } v_1 \text{ of } v.$$

The strongest postcondition does not say anything about termination of S. In this sense it is similar to the *weakest liberal precondition* $\mathrm{wlp}(S, Q)$ of [Di76], which does not require termination either. The strongest liberal postcondition used here is often denoted by $\mathrm{slp}(S, P)$, in analogy with the weakest liberal precondition. We prefer the notation $\mathrm{sp}(S, P)$, because it shows a symmetry with the corresponding notation for the weakest precondition.

The strongest postcondition for a statement $S: v$ satisfies the following properties, for arbitrary conditions $P, Q: v$:

$$\mathrm{sp}(S, \mathrm{false}) \Leftrightarrow \mathrm{false} \tag{5}$$

$$\forall v \cdot (P \Rightarrow Q) \Rightarrow (\mathrm{sp}(S, P) \Rightarrow \mathrm{sp}(S, Q)) \tag{6}$$

$$\mathrm{sp}(S, P \wedge Q) \Rightarrow \mathrm{sp}(S, P) \wedge \mathrm{sp}(S, Q) \tag{7}$$

$$\mathrm{sp}(S, P \vee Q) \Leftrightarrow \mathrm{sp}(S, P) \vee \mathrm{sp}(S, Q) \tag{8}$$

The last two rules generalize to arbitrary (even infinite) disjunctions and conjunctions.

The strongest postcondition is used to define partial correctness of program statements: statement S is *partially correct* with respect to precondition P and postcondition Q, denoted by $P\{S\}\,Q$, if $\mathrm{sp}(S, P) \Rightarrow Q$.

Relationship between wp *and* sp. Weakest precondition and strongest postcondition are related as follows. Let $S: v$ be a statement and $P, Q: v$ be conditions. We then have

$$\mathrm{sp}(S, \mathrm{wp}(S, Q)) \Rightarrow Q \tag{9}$$

$$\text{If } P \Rightarrow \mathrm{wp}(S, \mathrm{true}), \quad \text{then } P \Rightarrow \mathrm{wp}(S, \mathrm{sp}(S, P)) \tag{10}$$

The validity of these properties is seen as follows. Consider first property (9). Assume that the left hand side holds for state v_1. Then there is a state v_0 such that $v_1 \in S(v_0)$ and $\mathrm{wp}(S, Q)(v_0)$ holds. Hence, $Q(v_1')$ holds for any $v_1' \in S(v_0)$. In particular, $Q(v_1)$ holds, i.e. the right hand side holds.

For property (10) we argue as follows. Assume that $P \Rightarrow \mathrm{wp}(S, \mathrm{true})$. Assume further that $P(v_0)$ holds, for some arbitrary v_0. Then, by assumption, $\mathrm{wp}(S, \mathrm{true})(v_0)$ holds, so $\perp \notin S(v_0)$. For any v_1 in $S(v_0)$, $\mathrm{sp}(S, P)(v_1)$ holds. Hence $\mathrm{wp}(S, \mathrm{sp}(S, P))(v_0)$ holds. (These relationships between the weakest precondition and the strongest postconditions are also mentioned in [BeBi86].)

Frame Axiom. The following *frame axioms* for weakest preconditions and strongest postconditions are needed later. Let $S: v$ be a statement and let $P: v$ and $R: w$ be two predicates, where $v \cap w = \emptyset$. Then the weakest preconditions and

strongest postconditions for statement S: $\upsilon \cup w$, which in the sequel we write as S: υ, w, satisfies

$$\mathrm{wp}(S, P \wedge R) \Leftrightarrow \mathrm{wp}(S, P) \wedge R \tag{11}$$

$$\mathrm{sp}(S, P \wedge R) \Leftrightarrow \mathrm{sp}(S, P) \wedge R \tag{12}$$

By assumption, S can be restricted to the variables υ, so all variables of S are contained in υ. Hence, restricted to the variables υ and w, S: υ leaves the state component w unchanged. In other words, $(\upsilon_1, w_1) \in S(\upsilon_0, w_0)$ iff $\upsilon_1 \in S(\upsilon_0)$ and $w_1 = w_0$. The justification of the axioms is now straightforward, given this property.

3. Program Statements

We define the language of *program statements* by

$$\begin{array}{lll} S ::= \{B\} & & \textit{(assert statement)} \\ \quad | A & & \textit{(primitive statement)} \\ \quad | X & & \textit{(statement variable)} \\ \quad | S_1 ; S_2 & & \textit{(sequential composition)} \\ \quad | \textbf{if } B_1 \to S_1 \,[]\, B_2 \to S_2 \textbf{ fi} & & \textit{(conditional composition)} \\ \quad | \mu X \cdot S & & \textit{(recursion).} \end{array}$$

Here B, B_i are predicates, X is a statement variable and S, S_i are program statements.

The assert statement $\{B\}$ acts as a skip statement if condition B holds and as an abort statement otherwise. We write skip for $\{\mathrm{true}\}$ and abort for $\{\mathrm{false}\}$. The primitive atomic statements A are discussed below. The sequential and conditional compositions have their usual meaning, the latter being the guarded conditional statement. The construct $\mu X \cdot S(X)$ denotes recursion. The intended interpretation is that statement $S(X)$ is executed in such a way that whenever X is about to be executed, it is replaced by $S(X)$ and execution continues with this. The *iteration statement* is defined using recursion as

$$\textbf{do } B \to S \textbf{ od} = \mu X \cdot \textbf{if } B \to S;\ X \,[]\, \neg B \to \mathrm{skip} \textbf{ fi}.$$

The variables $\mathrm{var}(S)$ of a statement S are determined by the variables that occur in the primitive statements A, in the guards B_i, and the assert-statements $\{B\}$ of S. We assume that these are always known, so that we may determine whether a restriction of S to a given variable environment υ is permitted ($\mathrm{var}(S) \subseteq \upsilon$ must hold).

We will not give an explicit semantic definition of the meaning of statements here. Such a definition can be given in different ways, e.g. using denotational semantics [deB80] or operational semantics [Pl81]. It is sufficient for our pur-

pose to give the rules for computing the weakest precondition and strongest postcondition for each statement.

We assume that predicates are expressed in a language that permits infinite disjunctions, i.e. an infinitary logic like $L_{\omega_1\omega}$. The properties of this language are more or less those of ordinary first order logic, and its use simplifies the treatment of recursion. An overview of this logic is found in e.g. [Sc65], and its use in connection with weakest preconditions is described in [Ba80, Ba81a].

The theory developed here is independent of the class of primitive statements used in program statements, as well as of the class of guards in conditional statements and the assert statements. These syntactic categories are therefore not fixed here. We assume that the program variables that occur in these constructs are known. Furthermore, the rules for computing the weakest precondition and strongest postcondition for each primitive statement A: v and each predicate R: v are assumed to be given.

Let us assume first that the nondeterminism of the primitive statements is bounded. The definitions are extended to primitive statements of unbounded nondeterminism later. We define the *syntactic approximations* $S^n(X)$ of a formula $S(X)$ that contains statement variable X follows:

(i) $S^0(X) = X$
(ii) $S^{n+1}(X) = S(S^n(X))$, for $n = 0, 1, \ldots$.

Assume that the weakest precondition $\mathrm{wp}(A, R)$ is given for each primitive statement A. The weakest preconditions $\mathrm{wp}(S, R)$ for the statements S are then defined inductively by

1. $\mathrm{wp}(\{B\}, Q) = B \wedge Q$
2. $\mathrm{wp}(S_1; S_2, Q) = \mathrm{wp}(S_1, \mathrm{wp}(S_2, Q))$
3. $\mathrm{wp}(\textbf{if } B_1 \to S_1 \,[]\, B_2 \to S_2 \textbf{ if}, Q) =$
   $(B_1 \vee B_2) \wedge (B_1 \Rightarrow \mathrm{wp}(S_1, Q)) \wedge (B_2 \Rightarrow \mathrm{wp}(S_2, Q))$
4. $\mathrm{wp}(\mu X \cdot S(X), Q) = \bigvee_{n=0}^{\infty} \mathrm{wp}(S^n(\mathrm{abort}), Q)$.

Assume that the strongest postcondition $\mathrm{sp}(A, P)$ is given for each primitive statement A. The strongest postconditions $\mathrm{sp}(S, P)$ for the statements S are then defined inductively by

5. $\mathrm{sp}(\{B\}, P) = B \wedge P$
6. $\mathrm{sp}(S_1; S_2, P) = \mathrm{sp}(S_2, \mathrm{sp}(S_1, P))$
7. $\mathrm{sp}(\textbf{if } B_1 \to S_1 \,[]\, B_2 \to S_2 \textbf{ fi}, P) = \mathrm{sp}(S_1, B_1 \wedge P) \vee \mathrm{sp}(S_2, B_2 \wedge P)$
8. $\mathrm{sp}(\mu X \cdot S(X), P) = \bigvee_{n=0}^{\infty} \mathrm{sp}(S^n(\mathrm{abort}), P)$.

The rules for computing the weakest preconditions for the statements were originally formulated by Dijkstra [Di76], except the rule for recursion, which is due to Hehner [He79]. A treatment of weakest preconditions and strongest postconditions, both from a semantic and a proof theoretic point of view, is given by deBakker [deB80].

Even thought we do not make any specific assumptions about the primitive statement, we still need some primitive statements in our examples. For simplici-

ty, we use the multiple assignment statement $x := e$ as our primitive statement, where x is a list of distinct variables and e is a list of expressions. The weakest precondition and strongest postcondition for this construct are

$$\text{wp}(x := e, Q) = Q[e/x]$$
$$\text{sp}(x := e, P) = \exists y \cdot (P[y/x] \wedge x = e[y/x]).$$

We assume for simplicity that all functions in e are total, so that there is no need to consider possible undefined values in the expression.

3.1 Unbounded Nondeterminism

For simplicity, the definition of the weakest precondition for the recursive construct above assumes that the nondeterminism of primitive statements is bounded. If the nondeterminism of primitive statements is permitted to be unbounded, then the definition of the weakest precondition for the recursive construct above does not necessarily agree with the operational meaning that we want to assign to this construct. This has to do with termination: even if the recursive (or iterative) construct is guaranteed to terminate, no finite approximation of it need be guaranteed to terminate. Taking the disjunction only over the finite approximations does not therefore always give the right result.

We want to treat program specifications like program statements, so that there is only one kind of entity in the refinement calculus. In [Ba80] this is achieved by introducing a *nondeterministic assignment statement* $x := x' \cdot Q(x, x', y)$, where x and y are lists of program variables and x' is a list of fresh variables local to this statement. The effect of this statement is to assign to the variables x some values x' that make the condition $Q(x, x', y)$ true. If there is more than one list of values x' satisfying Q, the choice between these is nondeterministic. The statement aborts if no such list of values exists.

A specification consisting of a precondition $P(x, y)$ and a postcondition $Q(x, x', y)$ can be expressed as a nondeterministic assignment $x := x' \cdot (P(x, y) \wedge Q(x, x', y))$. For instance, if the purpose is to assign to x some new value that is larger than its previous value, this can be expressed by the statement $x := x' \cdot (x' \geqq x)$. If we assume that x ranges over the set of integers, then the nondeterminism of this statement is unbounded. A more detailed study of how specifications are integrated as part of the program language is outside the scope of this paper. We refer to [Ba80, Ba81a and Ba87] for further details.

In [Bo82], Boom shows how Dijkstra's definition of the weakest precondition for the while loop can be adapted to permit unbounded nondeterminism. We use the same idea below to adapt the weakest precondition for recursion so that unbounded nondeterminism is permitted. The basic idea is to take the infinite disjunction over the set of *all* ordinals, rather than over just the set of all *finite ordinals*, as is done in the definition above.

Define the predicates $H_\alpha(S, Q)$, where α is an ordinal, as follows:

(i) $H_0(S, Q) = \text{false}$

(ii) $H_{\alpha+1}(S, Q) = \mathrm{wp}(S(X_\alpha), Q)$, where $\mathrm{wp}(X_\alpha, R) = H_\alpha(S, R)$ for any R.
(iii) $H_\lambda(S, Q) = \bigvee_{\alpha<\lambda} H_\alpha(S, Q)$, when λ is a limit ordinal.

Here X_α are auxiliary statement variables, for every ordinal α. The weakest precondition of the recursive construct is then defined as

$$\mathrm{sp}(\mu X \cdot S(X), Q) = \bigvee_{\alpha \text{ an ordinal}} H_\alpha(S, Q).$$

Unbounded nondeterminism makes the weakest precondition transformer noncontinuous, as observed by Dijkstra [Di76]. For a recent discussion of unbounded nondeterminism along these lines and a reference to literature, see [ApPl86]. The effect of unbounded nondeterminism on the proof rule for loops is discussed also in [DiGa86]. Our results do not depend on the continuity of the predicate transformers, so in our case this does no harm.

4. Refinement between Statements

We are now ready to give a precise definition of a correct refinement. Basically a statement S is refined by a statement S' if S' preserves the total correctness of S: any pre-postcondition pair that S satisfies is also satisfied by S'. We will not, however, take this directly as a definition of correct refinement. Instead, we give an equivalent but simpler definition directly in terms of weakest preconditions. We first define refinement for statements that do not contain any statement variables.

Definition 1 (*Refinement*). Let $S: \upsilon$ and $S': \upsilon'$ be statements, where $\upsilon \subseteq \upsilon'$. Statement S is (correctly) refined by statement S', denoted by $S \leqq S'$, if for all postconditions $Q: \upsilon$

$$\mathrm{wp}(S, Q) \Rightarrow \mathrm{wp}(S', Q)$$

Note that the variable environment of the refining statement S' may include variables other than those in the variable environment of S, since υ may be a proper subset of υ'. This means that a refinement can introduce new auxiliary variables in order to carry out the required computation. However, each variable in the environment of S must also be in the environment of S'. (We discuss this issue in more detail in Sect. 7).

The fact that refinement preserves total correctness, and in fact is equivalent to the intuitive definition given in the introduction, is established by the following theorem.

Theorem 1. *Let $S: \upsilon$ and $S': \upsilon'$ be two statements, where $\upsilon \subseteq \upsilon'$. Then $S \leqq S'$ iff $P\langle S\rangle\, Q \Rightarrow P\langle S'\rangle\, Q$, for any $P, Q: \upsilon$.*

Proof. Assume that $S \leqq S'$. Assume that $P\langle S\rangle\, Q$, for some $P: \upsilon$ and $Q: \upsilon$, i.e. $P \Rightarrow \mathrm{wp}(S, Q)$. By the definition of refinement, $\mathrm{wp}(S, Q) \Rightarrow \mathrm{wp}(S', Q)$. Hence, $P \Rightarrow \mathrm{wp}(S', Q)$, i.e. $P\langle S'\rangle\, Q$ holds. For implication in the other direction, assume $P\langle S\rangle\, Q \Rightarrow P\langle S'\rangle\, Q$ holds for any pre- and postcondition $P, Q: \upsilon$. Assume that

$\mathrm{wp}(S, R)$ holds, for some R. This means that $\mathrm{wp}(S, R)\ \langle S\rangle\ R$ holds, so $\mathrm{wp}(S, R)\ \langle S'\rangle\ R$ also holds, by assumption. Hence, $\mathrm{wp}(S, R) \Rightarrow \mathrm{wp}(S', R)$, by the definition of total correctness. Therefore $S \leqq S'$, since R was arbitrarily chosen. Q.E.D.

The refinement relation is extended to statements with statement variables as follows.

Definition 2. Let $S(X)$: v and $S'(X)$: v', where $v \subseteq v'$, be program statements containing statement variable X. Then $S(X) \leqq S'(X)$ holds if $S(T)$: $v \leqq S'(T)$: v' for any substitution of a statement T for statement variable X in S and S', where $\mathrm{var}(T) \subseteq v$.

This definition generalizes in the obvious way to statements containing two or more statement variables. The following theorem establishes that refinement is a preorder, i.e. is reflexive and transitive.

Theorem 2. *Assume that v, v' and v'' are sets of program variables, satisfying $v \subseteq v' \subseteq v''$. Then the following conditions hold.*

(i) S: $v \leqq S$: v'.

(ii) S: $v \leqq S'$: v' *and* S': $v' \leqq S''$: v'' *implies* S: $v \leqq S''$: v''.

The straightforward proof is left to the reader. The first property is actually slightly stronger than reflexivity.

Refinement is not necessarily antisymmetric, so in general it is not a partial order. As usual, the preorder induces an equivalence relation between statements. The statements S: v and S': v are said to be *refinement equivalent*, denoted by $S \equiv S'$, if $S \leqq S'$ and $S' \leqq S$. From the definition of refinement, we see that two statements are refinement equivalent if and only if their weakest preconditions are equivalent for every postcondition Q: v. In other words, the two statements cannot be distinguished by weakest preconditions.

The assumption $v \subseteq v'$ in the definition of refinement is needed for transitivity, because it guarantees that any condition Q: v is also a condition on the program variables v', i.e. Q: v'. The following counter example shows that this condition cannot be omitted.

Let S: $\{x, z\}$ be the statement $x, z := x, 1$, S': $\{z\}$ be the statement $z := 1$ and S'': $\{x, z\}$ be the statement $x, z := 0, 1$. Then $S \leqq S'$, because

$$\mathrm{wp}(S, Q(x, z)) = Q(x, 1) = \mathrm{wp}(S', Q(x, z)),$$

for any $Q(x, z)$. Also $S' \leqq S''$ become

$$\mathrm{wp}(S', Q'(z)) = Q'(1) = \mathrm{wp}(S', Q'(z)),$$

for any $Q'(z)$. However, $S \leqq S''$ does not hold. Choosing e.g. $R(x, z) = (x = z)$, refinement would require that

$$\mathrm{wp}(S, R(x, z)) = (x = 1) \Rightarrow (0 = 1) = \mathrm{wp}(S'', R(x, z)),$$

for any value of x. This is, however, false for $x = 1$.

Digression. We can also define a notion of *partial refinement*, similar to refinement, but based on the strongest postcondition, as follows. Statement S: υ is partially refined by S': υ', denoted by $S \preceq S'$, is $\mathrm{sp}(S', P) \Rightarrow sp(S, P)$ for any precondition P: υ. Let us say that S and S' are *partially refinement equivalent,* denoted $S \approx S'$, if $S \preceq S'$ and $S' \preceq S$.

The refinement relation defined above and the partial refinement relation together characterize the semantics of a statement uniquely, as follows (these results are proved in [Ba81b]):

1. $S \leqq S' \preceq S$ iff S approximates S' in the Egli-Milner ordering [Pl76].
2. $S \equiv S' \approx S$ iff $S = S'$, in the sense of denoting the same mapping between initial and final states (i.e. $S(\upsilon_0) = S'(\upsilon_0)$ for each υ_0).

We do not need this unique characterization in this paper, since the coarser semantics determined by weakest preconditions alone is sufficient for studying total correctness of programs.

5. Monotonicity

We show now that the basic statement constructors, sequential and conditional composition, iteration and recursion, are monotonic with respect to the refinement relation.

Sequential Composition. Let S_i: υ and S_i': υ' be statements satisfying $S_i \leqq S_i'$ for $i = 1, 2$. We prove that $S_1; S_2$: $\upsilon \leqq S_1'; S_2'$: υ'. Let Q: υ be an arbitrary postcondition. Let $w = \upsilon' - \upsilon$. From $S_2 \leqq S_2'$ we have, explicitly quantifying over all program variables,

$$
\begin{aligned}
& S_2 \leqq S_2' \\
\Leftrightarrow & \{\text{By definition}\} \\
& \forall \upsilon, w \cdot (\mathrm{wp}(S_2, Q) \Rightarrow \mathrm{wp}(S_2', Q)) \\
\Leftrightarrow & \{w \text{ is not free in } \mathrm{wp}(S_2, Q)\} \\
(*) \quad & \forall \upsilon \cdot (\mathrm{wp}(S_2, Q) \Rightarrow \forall w \cdot \mathrm{wp}(S_2', Q))
\end{aligned}
$$

We now prove that $S_1; S_2$: $\upsilon \leqq S_1'; S_2'$: υ'.

$$
\begin{aligned}
& \mathrm{wp}(S_1; S_2, Q) \\
\Leftrightarrow & \{\text{By definition}\} \\
& \mathrm{wp}(S_1, \mathrm{wp}(S_2, Q)) \\
\Rightarrow & \{\text{Use rule of monotonicity (2) and } (*)\} \\
& \mathrm{wp}(S_1, \forall w \cdot \mathrm{wp}(S_2', Q)) \\
\Rightarrow & \{\text{Use } S_1 \leqq S_1' - \mathrm{wp}(S_2', Q) \text{ has only } \upsilon \text{ as free}\} \\
& \mathrm{wp}(S_1', \forall w \cdot \mathrm{wp}(S_2', Q)) \\
\Rightarrow & \{\text{Use (2)} - \forall w \cdot R \text{ implies } R \text{ is universally true}\} \\
& \mathrm{wp}(S_1', \mathrm{wp}(S_2', Q)) \\
\Leftrightarrow & \{\text{By definition}\} \\
& \mathrm{wp}(S_1'; S_2', Q).
\end{aligned}
$$

Conditional Composition. Let the assumptions be the same as in the previous case. We prove that

$$\textbf{if } B_1 \to S_1 \,[]\, B_2 \to S_2 \textbf{ fi}: \upsilon \leqq \textbf{if } B_1 \to S_1' \,[]\, B_2 \to S_2' \textbf{ fi}: \upsilon'.$$

The assumptions, together with the definition of refinement and property (2), give

$$\begin{aligned} &\mathrm{wp}(\textbf{if } B_1 \to S_1 \,[]\, B_2 \to S_2 \textbf{ fi}, Q) \\ &= (B_1 \vee B_2) \wedge (B_1 \Rightarrow \mathrm{wp}(S_1, Q)) \wedge (B_2 \Rightarrow \mathrm{wp}(S_2, Q)) \\ &\Rightarrow (B_1 \vee B_2) \wedge (B_1 \Rightarrow \mathrm{wp}(S_1', Q)) \wedge (B_2 \Rightarrow \mathrm{wp}(S_2', Q)) \\ &= \mathrm{wp}(\textbf{if } B_1 \to S_1' \,[]\, B_2 \to S_2' \textbf{ fi}, Q). \end{aligned}$$

Recursion. We prove that for any statements $S(X)$: υ and $T(X)$: υ', $S(X) \leqq T(X)$ implies $\mu X \cdot S(X)$: $\upsilon \leqq \mu X \cdot T(X)$: υ'.

Let us first establish an auxiliary result. We show that if S is monotone, then $S(X) \leqq T(X)$ implies $S^n(X) \leqq T^n(X)$, for any $n \geqq 0$. We prove this by induction. For $n = 0$ the result is trivial. Assume that the refinement holds for $n \geqq 0$. We then have, by the monotonicity of S, the induction hypothesis and the definition of refinement, that

$$S^{n+1}(X) = S(S^n(X)) \leqq S(T^n(X)) \leqq T(T^n(X)) = T^{n+1}(X),$$

which proves the case.

The required result is now proved as follows:

$$\begin{aligned} &S(X) \leqq T(X) \\ \Rightarrow &\{\text{By result above, choosing } X = \text{abort}\} \\ &S^n(\text{abort}) \leqq T^n(\text{abort}), \text{ for any } n \geqq 0 \\ \Leftrightarrow &\{\text{By definition of refinement}\} \\ &\mathrm{wp}(S^n(\text{abort}), Q) \Rightarrow \mathrm{wp}(T^n(\text{abort}), Q), \text{ for any } n \geqq 0, \text{ any } Q: \upsilon \\ \Rightarrow &\{\text{Property of infinite disjunction}\} \\ &\bigvee_{n=0}^{\infty} \mathrm{wp}(S^n(\text{abort}), Q) \Rightarrow \bigvee_{n=0}^{\infty} \mathrm{wp}(T^n(\text{abort}), Q), \text{ for any } Q: \upsilon \\ \Leftrightarrow &\{\text{Definition of weakest preconditions for recursion}\} \\ &\mathrm{wp}(\mu X \cdot S(X), Q) \Rightarrow \mathrm{wp}(\mu X \cdot T(X), Q), \text{ for any } Q: \upsilon \\ \Leftrightarrow &\{\text{Definition of refinement}\} \\ &\mu X \cdot S(X) \leqq \mu X \cdot T(X). \end{aligned}$$

Essentially the same proof goes through even if we permit the nondeterminism of the primitive statements to be unbounded. The only difference is that we have to use transfinite induction to establish the required result. We prove the monotonicity of the recursive construct for the case of unbounded nondeterminism in the appendix.

Iteration. The monotonicity of the iteration statement follows from the monotonicity of the above constructs. The iteration statement is defined as

$$\textbf{do } B \to S \textbf{ od} = \mu X \cdot \textbf{if } B \to S;\ X \,[]\, \neg B \to \text{skip } \textbf{fi}.$$

Assume that $S: v \leqq S': v'$. Since $X: v \leqq X: v'$, $S; X: v \leqq S'; X: v'$, by monotonicity of sequential composition. We then have

$$\textbf{if } B \to S;\ X \,[]\, \neg B \to \text{skip } \textbf{fi}:\ v \leqq \textbf{if } B \to S';\ X \,[]\, \neg B \to \text{skip } \textbf{fi}: v',$$

by monotonicity of conditional composition. Hence, by monotonicity of recursion we get the desired result,

$$\textbf{do } B \to S \textbf{ od}:\ v \leqq \textbf{do } B \to S' \textbf{ od}:\ v'.$$

We can now state the main result of this section.

Theorem 3 (Monotonicity). *Let $S[T]: v$ be a statement containing substatement $T: v$, and let $T': v'$ be some other statement, where $v \subseteq v'$. Replacing T by T' in S gives a statement $S[T']: v'$. Then $T \leqq T'$ implies $S[T] \leqq S[T']$.*

The proof of this is straightforward, and proceeds by induction on the structure of statements (mimicking the way in which iteration above is shown to be monotone). The theorem shows that it is always correct in a program derivation to replace a substatement with its refinement.

6. Reducing Refinement to Total Correctness

We show how refinement between two statements can be characterized by total correctness of the refining statement with respect to a specific pre- and postcondition that depend only on the statement being refined. This gives us a bridge between the refinement calculus, which works on the level of replacements of statement parts and relates statements to each other, and the weakest precondition calculus, which is concerned with the total correctness of specific statements and relates statements to specifications.

6.1 A Proof Rule for Refinement

The following theorem shows how to reduce refinement between two statements to a proof of a specific total correctness formula.

Theorem 4. *Let $S: v$ and $S': v'$ be two statements, where $v \subseteq v'$. Then $S \leqq S'$ holds if*

$$\text{wp}(S, \text{true}) \wedge v = v_0 \langle S' \rangle \text{ sp}(S, v = v_0), \tag{13}$$

where v_0 is some list of fresh program variables corresponding to the list of program variables v, $v \cap v_0 = \emptyset$.

Proof. Given (13) and $\text{wp}(S, Q)$, we have to prove $\text{wp}(S', Q)$. To begin, we prove $\text{sp}(S, v = v_0) \Rightarrow Q$:

$$\begin{aligned}
&\text{true}\\
\Leftrightarrow\ &\{v = v_0 \Rightarrow \text{true holds, and wp}(S, Q) \text{ holds by assumption}\}\\
&v = v_0 \Rightarrow \text{wp}(S, Q)\\
\Rightarrow\ &\{\text{Law of monotonicity (6)}\}\\
&\text{sp}(S, v = v_0) \Rightarrow \text{sp}(S, \text{wp}(S, Q))\\
\Rightarrow\ &\{\text{Use (9)}, \text{sp}(S, \text{wp}(S, Q)) \Rightarrow Q\}\\
&\text{sp}(S, v = v_0) \Rightarrow Q
\end{aligned}$$

Next, note that wp(S, true) holds, because wp(S, Q) holds by assumption. We now have:

$$
\begin{aligned}
& \text{true} \\
\Leftrightarrow & \{\text{Assume (13), use its definition}\} \\
& \text{wp}(S, \text{true}) \wedge v = v_0 \Rightarrow \text{wp}(S', \text{sp}(S, v = v_0)) \\
\Leftrightarrow & \{\text{wp}(S, \text{true}) = \text{true}\} \\
& v = v_0 \Rightarrow \text{wp}(S', \text{sp}(S, v = v_0)) \\
\Leftrightarrow & \{\text{Law of monotonicity (2), sp}(S, v = v_0) \Rightarrow Q, \text{ proved above}\} \\
& v = v_0 \Rightarrow \text{wp}(S', Q) \\
\Leftrightarrow & \{v, v_0 \text{ are unrestricted}\} \\
& \forall v, v_0 \cdot (v = v_0 \Rightarrow \text{wp}(S', Q)) \\
\Leftrightarrow & \{\text{Instantiation}\} \\
& \forall v \cdot (v = v \Rightarrow \text{wp}(S', Q)) \\
\Leftrightarrow & \{v = v \text{ is universally true, make quantification implicit}\} \\
& \text{wp}(S', Q). \quad \text{Q.E.D.}
\end{aligned}
$$

6.2 *A Characterization of Refinement*

Actually, the converse of the previous theorem is also true. This gives us the following characterization of refinement.

Theorem 5. *Let* S: v *and* S': v' *be two statements, where* $v \subseteq v'$. *Then* $S \leqq S'$ *holds if and only if*

$$\text{wp}(S, \text{true}) \wedge v = v_0 \langle S' \rangle \text{ sp}(S, v = v_0), \tag{14}$$

where v_0 *is some list of fresh program variables corresponding to the list of program variables* v, $v \cap v_0 = \emptyset$.

Proof. The previous theorem takes care of the if-part. To prove the other direction, assume that S: v and S': v' are two statements, $v \subseteq v'$, such that $S \leqq S'$. We want to prove that (14) holds. Because wp(S, true) $\wedge\, v = v_0 \Rightarrow$ wp(S, true), by property (10) we have

$$
\begin{aligned}
\text{wp}(S, \text{true}) \wedge v = v_0 & \Rightarrow \text{wp}(S, \text{sp}(S, \text{wp}(S, \text{true}) \wedge v = v_0)) \\
& \Rightarrow \text{wp}(S, \text{sp}(S, v = v_0)) \\
& \Rightarrow \text{wp}(S', \text{sp}(S, v = v_0)), \text{ by assumption,}
\end{aligned}
$$

thus establishing the required conclusion. Q.E.D.

This characterization gives us an intuitive interpretation of the refinement relation, as follows. The statement S: v is refined by statement S': v iff, for any initial assignment of values v_0 to the variables in v,

(i) if execution of S in this initial state is guaranteed to terminate, then execution of S' in this initial state is also guaranteed to terminate, and

(ii) any assignment of values to the variables v by S' is also a possible assignment of values to v by S.

More precisely, $S \leqq S'$ iff for any initial state v_0 of v, either $\perp \in S(v_0)$ or $S'(v_0) \subseteq S(v_0)$. Hence, if $S \leqq S'$, then both the domain of termination and the determinism of the result are non-decreasing when going from S to S'. When the variable environment of S' is an extension of the variable environment of S, then the same interpretation still holds, except that the values assigned by S' to variables not in the invironment of S are disregarded. (This relation turns out to be essentially equivalent to the Smyth ordering [Sm78] used in the context of denotational semantics to construct a power domain for bounded nondeterminism. The refinement relation as defined here was introduced in [Ba78] independently of the work be Smyth, and for a very different purpose).

7. Conservative Refinements

If statements S and S' are restricted to the same variable environment v, then we refer to $S \leqq S'$ as a *conservative refinement*. A conservative refinement does not introduce new auxiliary variables. We have the following result for conservative refinements.

Theorem 6. *Let z be the set of program variables that occur in S and S' and let v be any set of program variables, $z \subseteq v$. Then $S: v \leqq S': v$ iff $S: z \leqq S': z$.*

Proof. Assume that $S: z \leqq S': z$. Then

$$\mathrm{wp}(S, \mathrm{true}) \wedge z = z_0 \langle S' \rangle \mathrm{sp} \langle S, z = z_0),$$

by the characterization theorem for refinement. Assume now that

$$\mathrm{wp}(S, \mathrm{true}) \wedge z = z_0 \wedge y = y_0$$

holds, where $y = v - z$. This implies that

$$\mathrm{wp}(S', \mathrm{sp}(S, z = z_0)) \wedge y = y_0$$

holds, by the assumption. Since y does not occur in S', this is equivalent to

$$\mathrm{wp}(S', \mathrm{sp}(S, z = z_0) \wedge y = y_0),$$

by the frame axiom for weakest preconditions. Since y does not occur in S either, this is again equivalent to

$$\mathrm{wp}(S', \mathrm{sp}(S, z = z_0 \wedge y = y_0)),$$

by the frame axiom for strongest postconditions. This proves that

$$\mathrm{wp}(S, \mathrm{true}) \wedge z = z_0 \wedge y = y_0 \langle S' \rangle \mathrm{sp}(S, z = z_0 \wedge y = y_0).$$

Hence, $S: v \leqq S': v$, by the characterization theorem for refinement.

The implication in the other direction follows directly from the definition of refinement. Q.E.D.

The following result follows immediately from this theorem.

Corollary 1. *Let S and S' be program statements. Then $S: v \leqq S': v$ iff $S: w \leqq S': w$, for any sets of variables v and w that contain all variables of S and S'.*

Let us now consider simplifying the theory presented thus far by dropping the explicit indication of variable environments for statements. Let us make the following conventions. If no variable restriction is indicated for a program statement S or predicate R, then it is said to be *unrestricted* and is identified with the (restricted) statement S: Var or (restricted) predicate R: Var, respectively. In other words, S is considered a statement and R a predicate on *all* program variables. By the definition of refinement, $S \leqq S'$ holds for unrestricted statements iff

$$\mathrm{wp}(S, R) \Rightarrow \mathrm{wp}(S', R),$$

for any unrestricted predicate R.

Refinement between unrestricted statements is a special case of conservative refinement. Hence, we have the following alternative characterization of refinement between unrestricted statements.

Corollary 2. *Let S and S' be unrestricted statements. Then $S \leqq S'$ if f $S: z \leqq S': z$, where z is the set of variables that occur in S and S'.*

This result shows that to prove refinement between unrestricted statements by the proof rule of the previous section, one needs only take into account the initial values of the program variables that occur in S and S'.

In conclusion, we have the following result, which says that conservative refinement and refinement between unrestricted statements are equivalent notions:

Corollary 3. *The refinement $S \leqq S'$ holds for unrestricted statements iff $S: v \leqq S': v$ holds, for any set of variables v that includes all variables in S and S'.*

We could restrict attention to refinements between unrestricted statements in program derivations, thus ignoring the variable environment of statements altogether. This would simplify the theory of refinements, and would also make program derivations somewhat simpler, since one would not need to keep track of the variable environments. It would, however, have the disadvantage that no new auxiliary variables could be introduced by a refinement: If $S \leqq S'$ is a refinement between unrestricted program statements, then excecution of S' may assign new values only to those variables that are explicitly changed by S. If S' temporarily changes the values of some variables that do not occur in S, these variables must have their initial values restored on termination of S'. This is an awkward and unconventional way of using auxiliary variables. As we have shown above, it is also an unnecessary restriction. By explicitly keeping track of the variable environments of program statements, we are free to introduce new auxiliary variables when needed.

The refinements used in [Ba80] are conservative. The introduction of auxiliary variables is handled there by special primitive statements that can temporarily

add new variables to a state or delete variables from the state. This same approach could have been used here also, e.g. by adding blocks with local variable declarations. This approach is also taken in [Ba87]. The simple language considered here is, however, in some sense the backbone of any real programming language, and it was considered important to be able to capture the way in which program derivation is done in this language, including the use of auxiliary variables in derivations, without adding any unnecessary language constructs.

In practice, it is important to be able to introduce auxiliary variables in the course of program derivation. In most cases, the refinements go from abstract and conceptually simple statements to concrete and more efficient statements. Concretness and efficiency is usually achieved by the addition of new auxiliary variables, by which time is traded for space, or abstract entities are replaced by more concrete ones.

By distinguishing between different variable environments of program statements, we can combine conservative and non-conservative refinements during program derivation. The basic rule is to use conservative refinements throughout the derivation, except for those steps where auxiliary variables are explicitly needed. Since conservative refinements are equivalent to refinements between unrestricted variables, we need not consider the variable environments in conservative refinements. When an auxiliary variable is introduced, by a (non-conservative) refinement T: $x \leqq T'$: x', one only needs to check that the new variables in x' do not occur elsewhere in the statement $S[T]$ in which the replacement of T by T' is done. Thus, our approach supports and justified the way in which auxiliary variables are usually employed in program derivations.

8. Replacements in Context

We now consider replacements that are correct in the specific context in which they occur, but that are not correct in every context. For simplicity, we restrict ourselves here to refinements between unrestricted program statements. The results, however, hold for the general case also.

Assume that $S[T] \leqq S[T']$ holds but $T \leqq T'$ does not. Replacement cannot therefore be justified by monotonicity alone. We could in principle prove $S[T] \leqq S[T']$ directly, using the proof rule for refinement given in Sect. 6, but unless S is a very simple statement, this would be quite tedious. A way to handle this kind of replacement systematically, within the framework developed in the previous sections, is now described.

The context-dependent replacement is carried out in two steps:

(1) First prove $S[T] \leqq S[\{Q\}; T]$, for some assertion Q.

(2) Then prove $\{Q\}; T \leqq T'$.

By monotonicity, we then have $S[\{Q\}; T] \leqq S[T']$ and by transitivity, $S[T] \leqq S[T']$.

Intuitively, $S[T] \leqq S[\{Q\}; T]$ says that for any initial state in which S is guaranteed to terminate, during execution of S condition Q holds prior to executing T. If this was not the case, then abortion would occur, so the refinement

could not hold (note that $\{Q\}; T$ and T have the same effect whenever Q holds). Assertion $\{Q\}$ in $S[\{Q\};T]$ thus describes the context in which statement T is executed, and it is referred to as a *context assertion.* This context information is used in the second step, where we prove that $\{Q\}; T \leqq T'$. By the definition of refinement and the rules for computing weakest preconditions, this holds iff

$$Q \wedge \mathrm{wp}(T, R) \Rightarrow \mathrm{wp}(T', R),$$

for any postcondition R. Thus, the behaviour of T' is restricted only for initial states that satisfy Q. In this way, the context in which statement T is executed is taken into account in the replacement.

8.1 Context Introduction and Elimination Rules

We need rules for introducing and eliminating context assertions. Only one rule is needed for *elimination*:

$$\text{If } Q \Rightarrow Q', \quad \text{then } \{Q\} \leqq \{Q'\}.$$

Since $Q \Rightarrow$ true, this gives us $\{Q\} \leqq$ skip for any Q. Hence, $S[\{Q\}] \leqq S[\text{skip}] \equiv S$, by monotonicity.

Context assertions can be *introduced* using rules (C0)–(C6) below. Each rule has a conclusion of the form

$$\{P\}; S \equiv \{P\}; S[\{Q_1\}, \ldots, \{Q_n\}]. \tag{15}$$

This says that the context assertions $\{Q_1\}, \ldots, \{Q_n\}$ can be inserted at the specified places in S without affecting its total correctness provided P holds initially. Going the other way, the assertions can be removed from the right hand side, without affecting total correctness. Actually, (15) is equivalent to

$$\{P\}; S \leqq S[\{Q_1\}, \ldots, \{Q_n\}].$$

The latter implies $\{P\}; S \leqq \{P\}; S[\{Q_1\}, \ldots, \{Q_n\}, \ldots, \{Q_n\}]$ directly, by the definition of refinement, while refinement in the other direction follows by the rule for eliminating context assertions. Hence, in reasoning about context assertions below, we only consider this kind of refinement.

The rules for introducing context assertions are given below. They are basically adaptions of corresponding proof rules for partial correctness, except for the first rule, which has no counterpart in Hoare's logic.

C0. *Termination.* $S \equiv \{\mathrm{wp}(S, \text{true})\}; S$.

C1. *Consequence.* If $\{P\}; S \equiv \{P\}; S[\{Q\}]$, $P' \Rightarrow P$, and $Q \Rightarrow Q'$, then

$$\{P'\}; S \equiv \{P'\}; S[\{Q'\}].$$

C2. *Assert-statement.* $\{P\}; \{Q\} \equiv \{P\}; \{Q\}; \{P \wedge Q\}$.

C3. *Primitive statements.* We can propagate assertions forward and backward through primitive statement A:

$$\{P\}; A \equiv \{P\}; A; \{\mathrm{sp}(A, P)\} \quad \text{(forward propagation)}$$
$$A; \{Q\} \equiv \{\mathrm{wp}(A, Q)\}; A; \{Q\} \quad \text{(backward propagation)}.$$

C4. *Sequential composition.* If $\{P\}; S_1 \equiv \{P\}; S_1; \{Q\}$ and $\{Q\}; S_2 \equiv \{Q\}; S_2\ \{R\}$, then

$$\{P\}; S_1; S_2 \equiv \{P\}; S_1; \{Q\}; S_2; \{R\}$$

C5. *Conditional composition.* If $\{P \wedge B_i\}; S_i \equiv \{P \wedge B_i\}; S_i; \{Q\}$ for $i = 1, 2$, then

$$\{P\}; \textbf{if } B_1 \to S_1 \,[]\, B_2 \to S_2 \textbf{ fi}$$
$$\equiv \{P\}; \textbf{if } B_1 \to \{P \wedge B_1\}; S_1 \,[]\, B_2 \to \{P \wedge B_2\}; S_2 \textbf{ fi}; \{Q\}$$

C6. *Recursion.* If

$$\{P\}; X \leqq \{P\}; X; \{Q\} \Rightarrow \{P\}; S(X) \leqq \{P\}; S(\{P\}; X); \{Q\},$$

then

$$\{P\}; \mu X \cdot S(X) \equiv \{P\}; \mu X \cdot (\{P\}; S(\{P\}; X)); \{Q\}.$$

The last rule is somewhat complicated. Intuitively, it says that we may introduce context assertions $\{P\}$ and $\{Q\}$ into a recursive construct, $\{P\}$ at the beginning of the body and immediately before the recursive call and $\{Q\}$ at the end of the body, provided that the corresponding context introductions can be done in the body alone assuming that they are permitted for each recursive call.

These rules can be used either in a top-down or bottom-up manner to introduce assertions at appropriate places. Information about the domain of termination of a statement can be introduced by rule (C0) and propagated inwards by the other rules for context introduction. The validity of these rules is given by the following theorem.

Theorem 7. *The context assertion introduction rules* (C0)–(C6) *are valid.*

Proof. We prove here the validity for the forward propagation rule for primitive statements and the rule for recursion. The proof of the other cases are straightforward.

(C3) We should prove that $\{P\}; A \leqq A; \{\mathrm{sp}(A, P)\}$. Choose R arbitrarily. We have

$$\mathrm{wp}(\{P\}; A, R) = P \wedge \mathrm{wp}(A, R) \Rightarrow \mathrm{wp}(A, \mathrm{true}).$$

Hence, we can use rule (10) and (2), to get

$$\begin{aligned} P \wedge \mathrm{wp}(A, R) &\Rightarrow \mathrm{wp}(A, \mathrm{sp}(A, P \wedge \mathrm{wp}(A, R)) \wedge \mathrm{wp}(A, R) && \text{by (10)} \\ &\Rightarrow \mathrm{wp}(A, \mathrm{sp}(A, P)) \wedge \mathrm{wp}(A, R) && \text{by (2)} \\ &\Rightarrow \mathrm{wp}(A, \mathrm{sp}(A, P) \wedge R) \\ &\Rightarrow \mathrm{wp}(A; \{\mathrm{sp}(A, P)\}, R), \end{aligned}$$

which proves the case.

(C6) Assume, for all X,

$$\{P\}; X \leqq \{P\}; X; \{Q\}$$

implies

$$\{P\}; S(X) \leqq \{P\}; S(\{P\}; X); \{Q\}. \tag{16}$$

We should prove

$$\{P\}; \mu X \cdot S(X) \leqq \mu X \cdot (\{P\}; S(\{P\}; X)); \{Q\}. \tag{17}$$

Let us first prove that for each $i \geqq 0$,

$$\{P\}; S(S^i(\text{abort})) \leqq S(\{P\}; S^i(\text{abort})); \{Q\}.$$

For $i=0$, this follows from assumption (16), by noting that $\{P\}; \text{abort} \leqq \text{abort}; \{Q\}$. Assume that the property holds for $i \geqq 0$. We then have

$$\{P\}; S(S^i(\text{abort})) \leqq S(\{P\}; S^i(\text{abort})); \{Q\} \leqq S(S^i(\text{abort})); \{Q\},$$

i.e.

$$\{P\}; S^{i+1}(\text{abort}) \leqq S^{i+1}(\text{abort}); (Q\}.$$

Hence, by (16), we have the desired

$$\{P\}; S(S^{i+1}(\text{abort})) \leqq S(\{P\}; S^{i+1}(\text{abort})); \{Q\}.$$

Let $T(X) = \{P\}; S(\{P\}; X)$. We now show by induction that

$$\{P\}; S^i(\text{abort}) \leqq T^i(\text{abort}); \{Q\},$$

for each i. For $i=0$, the result is immediate. Assume the result holds for $i \geqq 0$. Then

$$\begin{aligned}
\{P\}; S^{i+1}(\text{abort}) &\equiv \{P\}; S(S^i(\text{abort})) \\
&\leqq \{P\}; S(\{P\}; S^i(\text{abort})); \{Q\}, \text{ by the assumption (16)} \\
&\leqq \{P\}; S(\{P\}; T^i(\text{abort}); \{Q\}); \{Q\}, \text{ induction hypothesis} \\
&\leqq \{P\}; S(\{P\}; T^i(\text{abort})); \{Q\} \\
&\equiv T^{i+1}(\text{abort}); \{Q\}.
\end{aligned}$$

Now choose an arbitrary R, and calculate the weakest precondition for the left hand side of (17):

$$\begin{aligned}
\text{wp}(\{P\}; \mu X \cdot S(X), R) &= P \wedge \bigvee_{i=0}^{\infty} \text{wp}(S^i(\text{abort}), R) \\
&\Leftrightarrow \bigvee_{i=0}^{\infty} (P \wedge \text{wp}(S^i(\text{abort}), R)) \\
&\Leftrightarrow \bigvee_{i=0}^{\infty} \text{wp}(\{P\}; S^i(\text{abort}), R) \\
&\Rightarrow \bigvee_{i=0}^{\infty} \text{wp}(T^i(\text{abort}); \{Q\}, R) \\
&\Rightarrow \bigvee_{i=0}^{\infty} \text{wp}(T^i(\text{abort}), \text{wp}(\{Q\}, R)) \\
&\Leftrightarrow \text{wp}(\mu X \cdot (\{P\}; S(\{P\}; X)), \text{wp}(\{Q\}, R)) \\
&\Leftrightarrow \text{wp}(\mu X \cdot (\{P\}; S(\{P\}; X)); \{Q\}, R)).
\end{aligned}$$

Thus the conclusion (17) is established. Q.E.D.

8.2 Example: Context Introduction in Iteration

As an example of using these rules, we show how to derive the following context introduction rule for the iteration statement:

C7. *Iteration.* If $\{P \wedge B\}; S \equiv \{P \wedge B\}; S; \{P\}$, then

$$\{P\}; \mathbf{do}\ B \to S\ \mathbf{od} \equiv \{P\}; \mathbf{do}\ B \to \{P \wedge B\}; S; \{P\}\ \mathbf{od}; \{P \wedge \neg B\}$$

We prove this as follows. Assume that

$$\{P \wedge B\}; S \equiv \{P \wedge B\}; S; \{P\}.$$

Let us further assume that

$$P; X \leqq X; P \wedge \neg B.$$

By rule (C4), this gives us

$$\{P \wedge B\}; S; X \equiv \{P \wedge B\}; S; X; \{X \wedge \neg B\}.$$

We also have, by rule (C2),

$$\{P \wedge \neg B\}; \mathbf{skip} \equiv \{P \wedge \neg B\}; \mathbf{skip}; \{P \wedge \neg B\}.$$

Using rule (C5), we get

$$\begin{aligned} &\{P\}; \mathbf{if}\ B \to S; X \,[]\, \neg B \to \mathbf{skip}\ \mathbf{fi} \\ \equiv &\{P\}; \mathbf{if}\ B \to \{P \wedge B\}; S; X \,[]\, \neg B \to \{P \wedge \neg B\}; \mathbf{skip}\ \mathbf{fi}; \{P \wedge \neg B\} \\ \leqq &\mathbf{if}\ B \to S; \{P\}; X \,[]\, \neg B \to \mathbf{skip}\ \mathbf{fi}; \{P \wedge \neg B\}. \end{aligned}$$

Hence,

$$\begin{aligned} &\{P\}; \mu X \cdot \mathbf{if}\ B \to S; X \,[]\, \neg B \to \mathbf{skip}\ \mathbf{fi} \\ \equiv &\{P\}; \mu X \cdot \{P\}; \mathbf{if}\ B \to S; \{P\}; X \,[]\, \neg B \to \mathbf{skip}\ \mathbf{fi}; \{P \wedge \neg B\}, \end{aligned} \tag{18}$$

by recursion rule (C6). On the other hand,

$$\begin{aligned} &\{P\}; \mathbf{if}\ B \to S; \{P\}; X \,[]\, \neg B \to \mathbf{skip}\ \mathbf{fi} \\ \leqq &\mathbf{if}\ B \to \{P \wedge B\}; S; \{P\}; X \,[]\, \neg B \to \mathbf{skip}\ \mathbf{fi}, \end{aligned}$$

so, by monotonicity of recursion,

$$\begin{aligned} &\mu X \cdot (\{P\}; \mathbf{if}\ B \to S; \{P\}; X \,[]\, \neg B \to \mathbf{skip}\ \mathbf{fi}) \\ \leqq &\mu X \cdot \mathbf{if}\ B \to \{P \wedge B\}; S; \{P\}; X \,[]\, \neg B \to \mathbf{skip}\ \mathbf{fi}. \end{aligned}$$

This, together with (18), yields

$$\begin{aligned} &\{P\}; \mu X \cdot \mathbf{if}\ B \to S; X \,[]\, \neg B \to \mathbf{skip}\ \mathbf{fi} \\ \leqq &\{P\}; \mu X \cdot \mathbf{if}\ B \to \{P \wedge B\}; S; \{P\}; X \,[]\, \neg B \to \mathbf{skip}\ \mathbf{fi}; \{P \wedge \neg B\}, \end{aligned}$$

i.e.

$$\{P\}; \mathbf{do}\ B \to S\ \mathbf{od} \leqq \{P\}; \mathbf{do}\ B \to \{P \wedge B\}; S; \{P\}, \mathbf{od}; \{P \wedge \neg B\}.$$

Refinement in the other direction follows immediately from the elimination rule for context assertions.

9. Correctness Proofs and Context Introduction

There is a rather close connection between being allowed to introduce a context assertion in a statement and partial and total correctness of that statement. Consider the special case $\{P\}; S \equiv \{P\}; S; \{Q\}$. We will prove that this is equivalent to

$$P \wedge \mathrm{wp}(S, \mathrm{true}) \Rightarrow \mathrm{wp}(S, Q). \tag{19}$$

This says that if P holds initially and if execution of S is guaranteed to terminate, then Q holds for the final state upon termination. We have:

$\{P\}; S \equiv \{P\}; S; \{Q\}$

$\Leftrightarrow$ {See discussion of (16)}

$\{P\}; S \leqq S; \{Q\}$

$\Leftrightarrow$ {Definition of refinement}

$\mathrm{wp}(\{P\}; S, R) \Rightarrow \mathrm{wp}(S; \{Q\}, R)$, for all R

$\Leftrightarrow$ {Definition of weakest preconditions}

$P \wedge \mathrm{wp}(S, R) \Rightarrow \mathrm{wp}(S, Q \wedge R)$, for all R

$\Leftrightarrow$ {Conjunction rule weakest preconditions}

$P \wedge \mathrm{wp}(S, R) \Rightarrow \mathrm{wp}(S, Q) \wedge \mathrm{wp}(S, R)$, for all R

$\Leftrightarrow$ {The second conjunct is redundant}

$P \wedge \mathrm{wp}(S, R) \Rightarrow \mathrm{wp}(S, Q)$, for all R

$\Leftrightarrow$ {Choosing $R = \mathrm{true}$, using $\mathrm{wp}(S, R) \Rightarrow \mathrm{wp}(S, \mathrm{true})$ for all R}

$P \wedge \mathrm{wp}(S, \mathrm{true}) \Rightarrow \mathrm{wp}(S, Q)$.

The relationship between context introduction and partial and total correctness is given by the following theorem.

Theorem 8. *Let S: v be a statement and P, Q: v be assertions. Then*

$$\{P\}; S \equiv \{P\}; S; \{Q\}$$

holds if $P\{S\}\,Q$ or $P\langle S\rangle\,Q$.

Proof. For partial correctness, the result is proved as follows. Assume that $P\{S\}Q$ holds, i.e. $\mathrm{sp}(S, P) \Rightarrow Q$. We have $P \wedge \mathrm{wp}(S, \mathrm{true}) \Rightarrow \mathrm{wp}(S, \mathrm{true})$. Hence, we can use (10), and get

$$\begin{aligned} P \wedge \mathrm{wp}(S, \mathrm{true}) &\Rightarrow \mathrm{wp}(S, \mathrm{sp}(S, P \wedge \mathrm{wp}(S, \mathrm{true}))) \\ &\Rightarrow \mathrm{wp}(S, \mathrm{sp}(S, P)) \\ &\Rightarrow \mathrm{wp}(S, Q); \end{aligned}$$

by the assumption, i.e. (19) holds. The result holds for total correctness, because total correctness implies partial correctness. Q.E.D.

In neither case does the converse hold in general, so context introduction is strictly weaker than partial and total correctness for nondeterministic statements. It coincides with partial correctness for deterministic statements. Choosing $S = \mathrm{abort}$ gives a counterexample for total correctness: We have $\{\mathrm{true}\}; \mathrm{abort} \leqq \mathrm{abort}; \{Q\}$, but $\mathrm{true} \Rightarrow \mathrm{wp}(\mathrm{abort}, Q)$ does not hold. Choosing $S =$ **if** true $\rightarrow$ skip $[]$ true $\rightarrow$ abort **fi** gives a counterexample for partial correctness. We have $\{P\}; S \leqq S; \{\neg P\}$, but $\mathrm{sp}(S, P) \Rightarrow \neg P$ does not hold.

This theorem allows us to use the results of a previous correctness proof when introducing context assertions in a statement. In particular, it allows us to use loop invariants directly as context assertions. Let us consider this in somewhat more detail.

Assume that we have a statement

$$S = S_0; \textbf{do } B \rightarrow S_1 \textbf{ od}$$

and that $P\langle S\rangle Q$ has been proved. The proof has been carried out by first proving

$$P\langle S_0\rangle P_1, \tag{20}$$

then proving that I is a loop invariant, i.e.

$$I \wedge B\langle S_1\rangle I, \tag{21}$$

from which we have deduced (having also proved termination of the loop) that

$$I\langle \textbf{do } B \rightarrow S_1 \textbf{ od}\rangle I \wedge \neg B,$$

and finally we have proved that

$$P_1 \Rightarrow I \quad \text{and} \quad I \wedge \neg B \Rightarrow Q.$$

From (21) we conclude by the theorem above that

$$\{I \wedge B\}; S_1 \equiv \{I \wedge B\}; S_1; \{I\}.$$

Hence, by rule (C7) for iteration, we conclude that

$$\{I\}; \textbf{do } B \rightarrow S_1 \textbf{ od} \equiv \{I\}; \textbf{do } B \rightarrow \{I \wedge B\}; S_1; \{I\}; \textbf{od}; \{I \wedge \neg B\}$$

Similarly, we conclude from (20) that

$$\{P\};S_0 \equiv \{P\};S_0;\{P_1\},$$

and, using consequence rule (C2), that

$$\{P\};S_0;\{P_1\} \equiv \{P\};S_0;\{I\}.$$

Thus we have

$$\begin{aligned} &\{P\};S_0;\textbf{do } B \rightarrow S_1 \textbf{ od} \\ \equiv\ &\{P\};S_0;\{I\};\textbf{do } B \rightarrow S_1 \textbf{ od} \\ \equiv\ &\{P\};S_0;\{I\};\textbf{do } B \rightarrow \{I \wedge B\};S_1;\{I\} \textbf{ od};\{I \wedge \neg B\} \end{aligned}$$

This shows how the proof of correctness allows us to introduce the assertions used in the proof as context assertions. What this means is that we may use the assertions in a *proof outline* of a statement as context assertions. Hence, any invariants established during proof of correctness of a statement can be used in the subsequent derivation steps as context information. This, of course, is just in line with the current practice of program transformations. It shows the double role of a correctness proof: on the one hand to establish that a statement does in fact satisfy its specification, on the other hand to provide a detailed analysis of how the statement works, thereby making subsequent changes to the statement easier.

Although a partial correctness assertion can always be introduced as a context assertion into a statement, the converse does not necessarily hold: there are context assertions that are not partial correctness assertions. The reason is that a context assertion need not hold when execution reaches it if there is another alternative execution path from the same initial state that does not terminate. If we restrict ourselves to programs for which termination is always guaranteed, then a context assertions may be introduced if and only if it is a partial correctness assertion.

10. An Example Derivation

We illustrate the refinement calculus by deriving a program. Let Fac be a statement that specifies the desired behaviour of a procedure for computing the factorial function. The possibility of overflow is explicitly taken into account in this specification:

$$\begin{aligned} \text{Fac: } &\textbf{if } n \geqq 0 \rightarrow \textbf{if } n! \leqq \max \rightarrow x := n! \\ &\qquad\qquad\quad\ \square\, n! > \max \rightarrow x := 0 \\ &\qquad\qquad\quad\ \textbf{fi} \\ &\textbf{fi} \end{aligned}$$

The variables of Fac are n, x and max, i.e. Fac: $\{\max, n, x\}$. Overflow is indicated by $x = 0$.

First Version. We implement Fac by Fac1: $\{x, n, \max, nn\}$

Fac1: **if** $n=0 \vee n=1 \rightarrow x:=1$
$\square n>1 \rightarrow x:=1; nn:=n;$
do $nn>1 \wedge x \neq 0 \rightarrow$ **if** $x*nn \leqq \max \rightarrow x:=x^*nn; nn:=nn-1;$
$\square x*nn>\max \rightarrow x:=0$
fi
od
fi

To prove that this implementation is correct, i.e. that Fac$\leqq$Fac1, we use the proof rule for refinement in Sect. 6 and prove

$$\mathrm{wp}(\mathrm{Fac}, \mathrm{true}) \wedge v=v_0 \langle \mathrm{Fac}\,1 \rangle \mathrm{sp}(\mathrm{Fac}, v=v_0),$$

where

$$\begin{aligned} \mathrm{wp}(\mathrm{Fac}, \mathrm{true}) \wedge v=v_0 &= x=x_0 \wedge n=n_0 \wedge \max=\max x_0 \wedge n \geqq 0 \\ \mathrm{sp}(\mathrm{Fac}, v=v_0) &= n \geqq 0 \wedge n=n_0 \wedge \max=\max x_0 \wedge \\ &\quad [(n! \leqq \max \wedge x=n!) \vee (n!>\max \wedge x=0)] \end{aligned}$$

This can be verified using the following loop invariant I:

$$\begin{aligned} I = {} & n=n_0 \wedge \max=\max_0 \wedge nn \geqq 1 \wedge \\ & [x*nn!=n_0! \vee (x=0 \wedge n_0!>\max)]. \end{aligned}$$

Second Version. We now observe that the test $x*nn \leqq \max$ will not really prevent overflow, because the test itself can cause an overflow. Hence, we replace it with something safer. We want to change the test to $x \leqq k$, where we have computed $k:=\max/nn$ immediately prior to the test. This, however, will be legal only if we know that $nn \neq 0$. Hence, we need to make a context-dependent replacement.

Because of the correctness proof, we know that preceding the loop body, the loop test holds, which means that context assertion $\{nn \geqq 1\}$ holds there also. Program Fac1 is thus refined by Fac2: $\{x, n, \max, nn\}$, in which $S1$ and $S2$ stand for the corresponding subparts in Fac1.

Fac2: **if** $n=0 \vee n=1 \rightarrow x:=1$
$\square n>1 \rightarrow x:=1; nn:=n;$
do $nn>1 \wedge x \neq 0 \rightarrow \{nn \geqq 1\};$
if $x*nn \leqq \max \rightarrow S1$
$\square x*nn>\max \rightarrow S2$
fi
od
fi

Third Version. Consider now the part $T1$: $\{x, n, \max, nn\}$ of Fac2:

$T1$: **if** $x*nn \leqq \max \rightarrow S1$
$\quad [] x*nn > \max \rightarrow S2$
$\quad$ **fi**

We replace $\{nn \geqq 1\}; T1$ by $T2$: $\{x, n, \max, nn, k\}$,

$T2$: $k := \max/nn;$
$\quad$ **if** $x \leqq k \rightarrow S1$
$\quad [] x > k \rightarrow S2$
$\quad$ **fi**

That this replacement is allowed, i.e. that $\{nn \geqq 1\}; T1 \leqq T2$, is seen as follows. Choose a postcondition Q (on variables $\max, n, x, nn$) arbitrarily. We have

$$\begin{aligned} \mathrm{wp}(\{nn \geqq 1\}; T1, Q) &= nn \geqq 1 \wedge (x*nn \leqq \max \Rightarrow \mathrm{wp}(S1, Q)) \wedge \\ &\quad (x*nn > \max \Rightarrow \mathrm{wp}(S2, Q)) \\ \mathrm{wp}(T2, Q) &= (x \leqq \max/nn \Rightarrow \mathrm{wp}(S1, Q)) \wedge \\ &\quad (x > \max/nn \Rightarrow \mathrm{wp}(S2, Q)) \end{aligned}$$

Obviously the latter is implied by the former. Making the replacement gives us the final program Fac3: $\{n, x, \max, nn, k\}$.

Fac3: **if** $n = 0 \vee n = 1 \rightarrow x := 1$
$\quad [] n > 1 \rightarrow x := 1; nn := n;$
$\quad\quad$ **do** $nn > 1 \wedge x \neq 0 \rightarrow k := \max/nn;$
$\quad\quad\quad\quad$ **if** $x \leqq k \rightarrow x := x*nn; nn := nn - 1;$
$\quad\quad\quad\quad [] x > k \rightarrow x := 0$
$\quad\quad\quad\quad$ **fi**
$\quad\quad$ **od**
$\quad$ **fi**

This program computes the factorial function as required by the initial specification, without the danger of overflow.

The Derivation Steps. Summarizing this derivation gives the following sequence of steps:

1. $\mathrm{Fac} \leqq \mathrm{Fac}1$ (proof rule for refinement)
2. $\mathrm{Fac}1 \equiv \mathrm{Fac}1\,[\{nn \geqq 1\}; T1]$ (context introduction rules)
3. $\{nn \geqq 1\}; T1 \leqq T2$ (definition of refinement)
4. $\mathrm{Fac}1\,[\{nn \geqq 1\}; T1] \leqq \mathrm{Fac}1\,[T2]$ (monotonicity of refinement)
5. $\mathrm{Fac} \leqq \mathrm{Fac}1\,[T2] = \mathrm{Fac}3$ (transitivity).

This derivation establishes that the final program Fac3 is a correct implementation of the original specification Fac. It illustrates how the refinement calculus allows us to describe the larger structure of a program derivation, and shows for each individual refinement step what justification is required for this step to be correct.

11. Concluding Remarks

The theory of correct refinements developed in this paper extends our earlier work [Ba78, Ba80, Ba81b] on the formalization of the stepwise refinement method. It provides a unified framework within which both the axiomatic, the stepwise refinement, and the transformational approach to program construction and verification can be described. The axiomatic approach, represented here by the weakest precondition calculus for total correctness, works on a lower level and relates statements to pre- and postconditions, while the stepwise refinement and transformational approach relate statements to each other. We have in this paper shown how these two levels are connected by the notion of correct refinement.

The main purpose of the paper has been to study the basic properties of the refinement relation. We have not considered how to apply this technique to specific methods for making refinement steps, and no effort has been made to show how this approach works in the derivation of large and/or complicated algorithms. In order to use this approach efficiently, we need to introduce a higher level language for program specifications. This is described in detail in [Ba80], where the refinement calculus is applied to procedural abstraction (implementing subprogram specifications), control abstraction (changing control structures) and data abstractions (changing the representation of program variables) in program derivation. The use of procedural abstraction is investigated further in an accompanying paper [Ba87]. The refinement calculus is applied to the formal derivation of a more intricate program, the marking algorithm in [BaMaRa], in a forthcoming paper. The technique is extended to handle derivations of distributed programs in another work currently in progress. The purpose of this paper has been to provide a foundation for these later developments and applications.

When finishing the final revision of this paper it became clear that two other people had started to work on the same notion of refinement that we describe here, Carroll Morgan [Morg86] and Joseph Morris [Morr87]. Seemingly unaware of the earlier work in [Ba78, Ba80], they introduce essentially the same kinds of concepts to describe stepwise refinement, including a form of the nondeterministic assignment statement for procedural abstraction. They derive a number of results that already were proved in our earlier work, such as monotonicity of program constructs and proof rules for handling the nondeterministic assignment statement. They also note that the refinement relation can be used to handle data refinement, a technique which is described in detail in [Ba78, Ba80]. However, both writers also extend some of our earlier results, e.g. by permitting full recursion with unbounded nondeterminism in the language, in a similar way as we have done here, and also introducing interesting new concepts, such as statements that permit miracles (i.e. do not satisfy the law (1).

Acknowledgements. I would like to thank David Gries for discussions and a (very) careful reading of the paper, Viking Hognas, Heikki Mannila and Kaisa Sere for valuable comments on preliminary versions of this paper, and Rod Burstall for some very illuminating questions. I also want to thank the two referees for their valuable comments on this paper.

Appendix: Monotonicity in Case of Unbounded Nondeterminism

We prove here that the recursive construct is monotonic even in the case that unbounded nondeterminism is permitted. For convenience, we repeat the definition of the weakest precondition here.

The predicates $H_\alpha(S, X, Q)$, where α is an ordinal, are defined as follows (we indicate explicitly the specific family X of auxiliary variables that is used):

(i) $H_0(S, X, Q) = \text{false}$
(ii) $H_{\alpha+1}(S, X, Q) = \text{wp}(S(X_\alpha), Q)$ where $\text{wp}(X_\alpha, R) = H_\alpha(S, X, R)$ for any R.
(iii) $H_\lambda(S, X, Q) = \bigvee_{\alpha<\lambda} H_\alpha(S, X, Q)$, when λ is a limit ordinal.

Here X_α are auxiliary statement variables, for every ordinal α. The weakest precondition of the recursive construct is then defined as

$$\text{wp}(\mu X \cdot S(X), Q) = \bigvee_{\alpha \text{ an ordinal}} H_\alpha(S, X, Q).$$

We should prove that $S(X) \leqq T(X)$ implies $\mu X \cdot S(X) \leqq \mu X \cdot T(X)$. We prove first by transfinite induction that $H_\alpha(S, X, Q) \rightarrow H_\alpha(T, Y, Q)$, for every ordinal α and every postcondition Q.

For $\alpha = 0$, the result is trivial. For $\alpha + 1$, we have that

$$H_{\alpha+1}(S, X, Q) = \text{wp}(S(X_\alpha), Q),$$

where $\text{wp}(X_\alpha, R) = H_\alpha(S, X, R)$ for any R. But, by the induction hypothesis, we have that $H_\alpha(S, X, R) \Rightarrow H_\alpha(T, Y, R)$, for any R, i.e. $\text{wp}(X_\alpha, R) \Rightarrow \text{wp}(Y_\alpha, R)$ for any R. This means that $X_\alpha \leqq Y_\alpha$. Thus, we have by monotonicity of S that $S(X_\alpha) \leqq S(Y_\alpha)$, so we get

$$\text{wp}(S(X_\alpha), Q) \Rightarrow \text{wp}(S(Y_\alpha), Q).$$

On the other hand, $S(X) \leqq T(X)$ by assumption, so we have

$$\text{wp}(S(Y_\alpha), Q) \Rightarrow \text{wp}(T(Y_\alpha), Q).$$

Altogether, $H_{\alpha+1}(S, X, Q) \Rightarrow H_{\alpha+1}(T, Y, Q)$ as required.

For a limit ordinal λ, we have

$$H_\lambda(S, X, Q) = \bigvee_{\alpha<\lambda} H_\alpha(S, X, Q).$$

By the induction assumption, $H_\alpha(S, X, Q) \Rightarrow H_\alpha(T, Y, Q)$ for every ordinal $\alpha < \lambda$, so

$$H_\alpha(S, X, Q) \Rightarrow \bigvee_{\alpha<\lambda} H_\alpha(T, Y, Q),$$

for every $\alpha < \lambda$. This gives us

$$\bigvee_{\alpha<\lambda} H_\alpha(S, X, Q) \Rightarrow \bigvee_{\alpha<\lambda} (H_\alpha(T, Y, Q),$$

i.e. the result also holds for the limit ordinal λ. Thus the result holds for every ordinal α.

Essentially the same argument as for the limit ordinal now gives us the main result. We have $H_\alpha(S, X, Q) \Rightarrow H_\alpha(T, Y, Q)$ for every ordinal α. Hence,

$$H_\alpha(S, X, Q) \Rightarrow \bigvee_{\alpha \text{ ordinal}} H_\alpha(T, Y, Q),$$

for every ordinal α. Hence,

$$\bigvee_{\alpha \text{ ordinal}} H_\alpha(S, X, Q) \Rightarrow \bigvee_{\alpha \text{ ordinal}} H_\alpha(T, Y, Q),$$

i.e.

$$\mu X \cdot S(X) \leqq \mu X \cdot T(X).$$

As seen from this proof, the argumentation is very similar to the argumentation in the case of bounded nondeterminism, the difference being mainly that in the latter case, the proof is only carried to the first limit ordinal ω. However, the basic arguments in the proof hold for any ordinal, as shown above.

References

[ApPl86] Apt, K.R., Plotkin, G.D.: Countable nondeterminism and random assignment. J. ACM **33** (4) 724–767 (1986)

[Ba78] Back, R.J.R.: On the correctness of refinement steps in program development (Ph.D. thesis). Report A-1978-4, Dept. of Computer Science, University of Helsinki, 1978

[Ba80] Back, R.J.R.: Correctness preserving program refinements: proof theory and applications. Mathematical Center Tracts 131, Mathematical Centre, Amsterdam 1980

[Ba81a] Back, R.J.R.: Proving total correctness of nondeterministic programs in infinitary logic. Acta Informatica **15** 233–250 (1981)

[Ba81b] Back, R.J.R.: On correct refinement of programs. J. Comput. Syst. Sci. **23** (1), 49–68 (1981)

[BaMaRa83] Back, R.J.R., Mannila, H., Raiha, K.J.: Derivation of efficient dag marking algorithms. ACM Conference on Principles of Programming Languages, Austin, Texas 1983

[Ba87] Back, R.J.R.: Procedural abstraction in the refinement calculus. Reports on Computer Science and Mathematics no. 55, 1987, Abo Akademi

[Ba88] Back, R.J.R.: Derivation of a dag marking algorithm in the refinement calculus (in preparation)

[deB80] deBakker, J.: Mathematical theory of program correctness, Englewood Cliffs: Prentice-Hall 1980

[BBPPW79] Bauer, F.L., Broy, M., Partsch, H., Pepper, P., Wossner, H.: Systematics of transformation rules. In: Bauer, F.L., Broy, M. (eds.) Program construction. (Lect. Notes Comput. Sci., Vol. 69) Berlin Heidelberg New York: Springer 1979

[BeBi86] Berlioux, P., Bizard, P.: Algorithms; the construction, proof and analysis of programs. New York: Wiley 1986

[Bo82] Boom, H.J.: A weaker precondition for loops. TOPLAS **4** (4), 668–677 (1982)

[BrPeWi80] Broy, M., Pepper, P., Wirsing, M.: On relations between programs. In: Robinet, B. (ed.). International Symposium on Programming. (Lect. Notes. Comput. Sci., Vol. 83, pp. 59–78) New York: Springer 1980

[BuDa771] Burstall, R.M., Darlington, J.: Some transformations for developing recursive programs. J. ACM **24** (1) 44–67 (1977)

[Di71] Dijkstra, E.W.: Notes on structured programming. In: Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R. (eds.) Structured programming. New York London: Academic Press 1971

[Di76] Dijkstra, E.W.: A discipline of programming. Englewood Cliffs: Prentice Hall 1976

[DiGa86] Dijkstra, E.W., Gasteren, A.J.M.: A simple fixpoint argument without the restriction to continuity. Acta Informatica **23** 1–7 (1986)

[Gr81] Gries, D.: The science of programming. Berlin Heidelberg New York: Springer 1981

[He79] Hehner, E.: Do considered od: a contribution to the programming calculus. Acta Informatica **11**, 287–304 (1979)

[He84] Hehner, E.: The logic of programming. Englewood Cliffs: Prentice-Hall 1984

[He84b] Hehner, E.: Predicative programming, part I. CACM **27** (2) 134–143 (1984)

[Ho69] Hoare, C.A.R.: An axiomatic basis for computer programming. CACM **12** (10) 576–580 (1969)

[Ho71] Hoare, C.A.R.: Proof of a program: FIND. CACM **14**, 39–45 (1971)

[Ho85] Hoare, C.A.R.: Programs are predicates. In: Hoare, C.A.R., Shepherdson, J.C. (eds.) Mathematical logic and programming languages. pp. 141–155. Englewood Cliffs: Prentice-Hall 1985

[Morg86] Morgan, C.: The specification statement. Manuscript 1986

[Morr87] Morris, J.: A theoretical basis for stepwise refinement and the programming calculus. Sci. Comput. Programming **9** 287–306 (1987)

[JaGr85] Jacobs, D., Gries, D.: General correctness. A unification of partial and total correctness. Acta Informatica **22** (1) 67–84 (1985)

[PaST83] Partsch, H., Steinbrugge, R.: Program transformation systems. ACM Comput. Surv. **15**, 199–236 (1983)

[Pl76] Plotkin, G.D.: A powerdomain construction. SIAM J. Comput. **5** (3) 452–487 (1976)

[Pl81] Plotkin, G.D.: Structural approach to operational semantics. Tech. report DAIMI FN-19, Comp. Science Department, Aarhus University, 1981

[Re81] Reynolds, J.C.: The craft of programming. Englewood Cliffs: Prentice-Hall 1981

[Sc65] Scott, D.: Logic with denumerably long formulas and finite strings of quantifiers. In: Addison, J., Henkin, L., Tarski, A. (eds.) Symposium on the Theory of Models. North-Holland 1965, 329–341

[Sm78] Smyth, M.B.: Power domains. J. Comput. Syst. Sci. **16**, 23–36 (1978)

[Wi71] Wirth, N.: Program development by stepwise refinement. CACM **14** 221–227 (1971)

Received August 17, 1987/March 21, 1988