Acta Informatica 27, 583–625 (1990)

Acta Informatica
© Springer-Verlag 1990

Duality in Specification Languages: A Lattice-theoretical Approach

R.J.R. Back[1] and J. von Wright[2]
[1] Åbo Akademi, Department of Computer Science, Lemminkäisenkatu 14, SF-20520 Turku, Finland
[2] Swedish School of Economics and Business Education, Biblioteksgatan 16, SF-65100 Vasa, Finland

Received June 5, 1989/November 16, 1989

Summary. A very general lattice-based language of commands, based on the primitive operations of substitution and test for equality, is constructed. This base language permits unbounded nondeterminism, demonic and angelic nondeterminism. A dual language permitting miracles is constructed. Combining these two languages yields an extended base language which is complete, in the sense that all monotonic predicate tranformers can be constructed in it. The extended base language provides a unifying framework for various specification languages; we show how two Dijkstra-style specification languages can be embedded in it.

1. Introduction

The weakest precondition calculus of Dijkstra [9] identifies the meaning of a program statement with its weakest precondition predicate transformer. Dijkstras "healthiness conditions" state that these predicate transformers for executable program statements are strict (they satisfy Dijkstra's "Law of Excluded Miracle"), monotonic, conjunctive (and-distributive) and continuous (or-continuous).

Extensions to the Dijkstra-style programming language that drop some of the healthiness conditions have subsequently been used to allow treatment of specifications, parallel programs and data refinement. Back [1, 2] introduces weakest preconditions for specifications. Unbounded nondeterminism is permitted, thus dropping the continuity condition. Further, a refinement relation on statements is introduced, such that S is refined by S' if

$$\forall Q \cdot (\mathrm{wp}_S(Q) \Rightarrow \mathrm{wp}_{S'}(Q))$$

where wp_S is the weakest precondition predicate transformer for S. The refinement relation is a preorder, essentially the same as the Smyth ordering of [23]. In Hehner [14], Morris [20] statements are identified with predicate transformers, an approach which we follow in this paper. Thus we write $S(Q)$ for $\mathrm{wp}_S(Q)$.

Offprint requests to: J. von Wright

In de Bakker [6], Nelson [21] the weakest precondition calculus is extended to cover partial state transformers, i.e. nonstrict statements. Morgan [19] and Morris [20] write specifications as pre-postcondition pairs, thus also allowing miraculous (nonstrict) statements. The actions of Back [4], used to model parallel programs, are also nonstrict. In Gardiner and Morgan [11] a conjunction operation on statements is defined, such that the resulting statements are not conjunctive. The same holds for the angelic basic statement of Back [5]. Generalized specification languages are also considered in Hesselink [15]. Thus, in going from a pure programming language to a specification language, we see that most of the original healthiness conditions have been questioned, in order to gain expressive power. In this sense a specification language is truly more general than a programming language, for which all the original healthiness conditions are well motivated.

In [20] it is noted that the monotonic predicate transformers form a lattice, with the partial order corresponding to the refinement ordering of statements. This is indirectly noted also in [11], where the operators $\square$ (choice) and $+$ (conjunction) on statements correspond to the lattice operators meet and join on predicate transformers. Similar operators, written $\cap$ and $\cup$, are introduced in Hoare et al. [16].

Nondeterminism in weakest precondition semantics has generally been assumed to be resolved demonically [9]. Angelic nondeterminism has been considered in Jacobs and Gries [17] (a relational model of weakest preconditions) and more generally in Broy [8].

In this paper we show how basic specification languages can be constructed within the complete lattice of monotonic predicate transformers, using only very simple primitive statements and functional (sequential) composition in addition to meets and joins. The languages permit demonic and angelic nondeterminism as well as mixtures of these.

The rest of the paper is as follows. In Sect. 2 we present general lattice-theoretic concepts and in Sect. 3 some results on lattices of functions on lattices. Section 4 contains definitions and results concerning predicates and predicate transformers. In Sect. 5 we define a lattice-based command language $\mathscr{C}^{\perp}$ of strict monotonic predicate transformers. Section 6 describes a dual language $\mathscr{C}^{\top}$ and an extended language $\mathscr{C}$ which is a combination of $\mathscr{C}^{\perp}$ and its dual. In Sect. 7 we prove completeness results. In Sect. 8 we consider a more conventional Dijkstra-style specification language with assignments, sequential and conditional composition, blocks and recursion but permitting demonic as well as angelic nondeterminism (the nondeterminism may be unbounded). We show how this language can be constructed within $\mathscr{C}^{\perp}$. In Sect. 9 we show how a more general Dijkstra-style language, which permits miracles and which has a self-dual nature, can be constructed within $\mathscr{C}$. Proofs of the more important results are included in the text, the other proofs can be found in the Appendix.

2. Partial Orders and Lattices

In this section, we present the general mathematical concepts of partial orders and lattices that will be needed in the subsequent sections. The presentation is quite short; proofs of most of the results related here can be found in [7, 12].

2.1. Partial Orders

Definition of partial order. A *partial order* on a set L is a binary relation $\leqq$ ("less-or-equal") on L that is reflexive, transitive and antisymmetric. If $\leqq$ is a partial order on L we say that the pair $(L, \leqq)$ is a *partially ordered set.* When there is no danger of ambiguity we simply say that L is a partially ordered set. If x and y are elements of the partially ordered set L, we write $x<y$ when $x \leqq y$ but $x \neq y$.

Lower and upper bounds. Assume that $(L, \leqq)$ is a partially ordered set and that K is a subset of L. An element $b \in L$ is a *lower bound* of K if $b \leqq x$ for every $x \in K$. The element b is a *greatest lower bound* of K, written

$$b = \bigwedge K$$

if b is a lower bound of K and $b' \leqq b$ for all lower bounds b' of K. If x and y are elements of L we write $x \wedge y$ for $\bigwedge \{x, y\}$.

Dually to the previous definition we define *upper bound* and *least upper bound* for a subset K and L. Thus b is an upper bound of K if $x \leqq b$ for every $x \in K$, and b is the least upper bound of K if b is an upper bound of K and $b \leqq b'$ for all upper bounds b' of K. The least upper bound of K is written

$$b = \bigvee K$$

and we write $x \vee y$ for $\bigvee \{x, y\}$.

2.2. Lattices

Definition of lattice. A partially ordered set $(L, \leqq)$ (or, for simplicity, just L) is called a *lattice* if $x \wedge y$ and $x \vee y$ exist for all pairs (x, y) of elements of L. In a lattice, the operator $\wedge$ is called the *meet* operator and the operator $\vee$ is called the *join* operator.

Defining properties of $\wedge$ *and* $\vee$. Assume that L is a lattice. Then the following four properties hold for arbitrary element a, b and c of L:

$$\begin{array}{lll}
a \wedge a = a, & a \vee a = a & (\textit{Idempotency}) \\
a \wedge b = b \wedge a, & a \vee b = b \vee a & (\textit{Commutativity}) \\
a \wedge (b \wedge c) = (a \wedge b) \wedge c, & a \vee (b \vee c) = (a \vee b) \vee c & (\textit{Associativity}) \\
a \wedge (a \vee b) = a, & a \vee (a \wedge b) = a & (\textit{Absorption})
\end{array}$$

Furthermore, any operators $\wedge$ and $\vee$ on an arbitrary set L, satisfying these four properties, define a lattice, and the corresponding partial order $\leqq$ is defined by

$$x \leqq y \overset{\text{def}}{=} x \wedge y = x$$

or, equivalently, by

$$x \leqq y \overset{\text{def}}{=} x \vee y = y,$$

Monotonicity of $\wedge$ *and* $\vee$. Both operators $\wedge$ and $\vee$ are monotonic in their arguments, i.e. if $x_1 \leqq y_1$ and $x_2 \leqq y_2$ then $x_1 \wedge x_2 \leqq y_1 \wedge y_2$ and $x_1 \vee x_2 \leqq y_1 \vee y_2$. The same holds for arbitrary (finite or infinite) meets and joins.

2.3. *Special Classes of Lattices*

Complete lattices, bottom and top elements. A lattice L is *complete* if $\bigwedge K$ and $\bigvee K$ exist for all subsets $K \subseteq L$ (we define $\bigwedge \emptyset = \bigvee L$ and $\bigvee \emptyset = \bigwedge L$). Every complete lattice contains a largest element, called the *top element*, which is denoted $\top$, and a smallest element called the *bottom element* and denoted $\bot$. We note that

$$\bot = \bigwedge L = \bigvee \emptyset,$$
$$\top = \bigvee L = \bigwedge \emptyset.$$

Distributive lattices. A lattice L is *distributive* if for any elements x, y and z of L the following two identities hold:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z),$$
$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z).$$

Note that these identities do not automatically generalize to infinite meets and joins; we call the corresponding identities for infinite meets and joins the *meet infinite distributivity condition* (MID) and the *join infinite distributivity condition* (JID). These conditions are defined as

$$x \vee (\bigwedge_{i \in I} x_i) = \bigwedge_{i \in I} (x \vee x_i) \qquad \text{(MID)},$$
$$x \wedge (\bigvee_{i \in I} x_i) = \bigvee_{i \in I} (x \wedge x_i) \qquad \text{(JID)}$$

where x and x_i, $i \in I$ are arbitrary elements of the lattice L, and I is any index set.

Boolean lattices, inverse elements. A complete distributive lattice B is called *boolean* if every element x has a unique *inverse element* x^{-1} such that $x \wedge x^{-1} = \bot$ and $x \vee x^{-1} = \top$. An important result is the following: *Every complete boolean lattice satisfies the infinite distributivity conditions* (MID) *and* (JID).

In a complete boolean lattice the following rules also hold (the *DeMorgan rules for complete boolean lattices*) for any index set I:

$$(\bigwedge_{i \in I} x_i)^{-1} = \bigvee_{i \in I} (x_i^{-1}) \qquad (1)$$

$$(\bigvee_{i \in I} x_i)^{-1} = \bigwedge_{i \in I} (x_i^{-1}) \qquad (2)$$

2.4. Sublattices

A subset K of a lattice L is a *sublattice* of L if K is closed with respect to meets and joins, i.e. if $x \wedge y \in K$ and $x \vee y \in K$ whenever $x, y \in K$. Thus K is a sublattice of L if and only if K is a lattice and all nonempty finite meets and joins in K yield the same result as when taken in L. We note that any sublattice of a distributive lattice is distributive.

Complete sublattices. A subset K of the lattice L is a *complete sublattice* of L if all meets and joins of subsets of K are in K. Thus it is not enough that K is a sublattice that is complete by itself, but all meets and joins in K must yield the same result as when taken in L.

3. Function Lattices

In this section, we consider the set of all total functions from arbitrary sets to lattices and the set of all total functions on a lattice.

3.1. The Lattice $[K \to L]$ of Functions from a Set to a Lattice

Let L be a lattice, K be any non-empty set, and let $[K \to L]$ be the set of all total functions from K to L. We introduce on $[K \to L]$ a partial order, called the *pointwise extension* of the partial order on L, by

$$f \leqq g \overset{\text{def}}{=} \forall x \cdot (f(x) \leqq g(x))$$

for any functions f and g from K to L. This makes $[K \to L]$ a lattice which is complete (distributive, boolean) if and only if L is complete (distributive, boolean). The definition of the partial order on $[K \to L]$ implies that meets and joins in $[K \to L]$ also are defined by pointwise extension from L.

3.2. Sublattices of the Lattice $[L \to L]$

We now consider two sublattices of the lattice $[L \to L]$ where L is a lattice.

The monotonic functions. A function f on L is *monotonic* if

$$\forall x, y \in L \cdot (x \leqq y \Rightarrow f(x) \leqq f(y)).$$

The monotonic functions form a sublattice of $[L \to L]$, denoted $[L \to L]_M$. Meets and joins in $[L \to L]_M$ thus yield the same result as when taken in $[L \to L]$. Furthermore, if L is complete, then $[L \to L]_M$ is complete and has the same bottom and top elements as $[L \to L]$.

The strict monotonic functions. We now assume that L is a complete lattice. A function f on L is *$\perp$-strict* if it maps $\perp_L$ to $\perp_L$. The $\perp$-strict functions form a sublattice of $[L \to L]$. We will not consider this sublattice but instead consider the functions that are both $\bot$-strict and monotonic. We denote this sublattice $[L \to L]_{SM}$. The bottom element of $[L \to L]_{SM}$ is the same as that of $[L \to L]$. The top element, however, is different from that of $[L \to L]$. It is the function that maps $\bot_L$ to $\bot_L$ and all other elements of L to $\top_L$.

If L is a complete boolean lattice then $[L \to L]$ is also complete and boolean. However, neither $[L \to L]_M$ nor $[L \to L]_{SM}$ is boolean. To see this, take any function that maps $\bot$ to $\bot$ and $\top$ to $\top$. Its inverse would have to map $\bot$ to $\top$ and $\top$ to $\bot$, thus being neither monotonic nor strict.

However we have the following important result: *The lattices* $[L \to L]_M$ *and* $[L \to L]_{SM}$ *satisfy the conditions* (MID) *and* (JID). This holds because both are sublattices of the complete boolean lattice $[L \to L]$ and all nonempty meets and joins give the same result as in $[L \to L]$.

We summarize the important results concerning $[L \to L]_{SM}$ in the following theorem:

Theorem 1. *The lattice $[L \to L]_{SM}$ is a distributive sublattice of $[L \to L]$. As a lattice, $[L \to L]_{SM}$ is complete and all meets and joins, except $\bigwedge\emptyset$, yield the same result in $[L \to L]_{SM}$ as in $[L \to L]$. In particular, $\bot_{[L \to L]_{SM}} = \bot_{[L \to L]}$ and the identities* (MID) *and* (JID) *hold in $[L \to L]_{SM}$.*

3.3. *Fixpoints*

Let f be a function on L. An element x of L is a *fixpoint* of f if $f(x) = x$. If x is less than all other fixpoints of f we say that x is the *least fixpoint* of f. We will need the following result, known as the Knaster-Tarski fixpoint theorem [24]: *Every monotonic function on a complete lattice has a complete lattice of fixpoints.* In particular, every monotonic function on a complete lattice has a least and a greatest fixpoint.

4. The Lattice of Predicate Transformers

In this section, we define predicate transformers by repeatedly using the function lattice construction.

4.1. *The Lattice Bool*

Let *Bool* be the set of truth values for a two-valued logic, $Bool = \{ff, tt\}$. We order the two (distinct) elements of *Bool* so that $ff < tt$. Thus the partial ordering relation corresponds to logical implication. This makes *Bool* a complete boolean lattice where $\wedge$ and $\vee$ have their usual meaning as logical connectives. In *Bool*, ff is the bottom element and tt is the top element. The two elements are the inverses of each other.

We give the connectives $\neg$ and $\Rightarrow$ their usual meanings in *Bool*, i.e. $\neg x$ is the inverse of x and $x \Rightarrow y$ is an abbreviation for $\neg x \vee y$.

4.2. *Variables, Values and the State Space*

We assume *Var* is a countable set of *program variables.* A typical element of *Var* is denoted u while v denotes a typical list of *distinct* elements of *Var* (lists can be finite or infinite). Sometimes we will need to consider all the elements of *Var* as a list, which we will then denote V.

We further assume a nonempty set D of values, with typical element c and typical list of elements d. The length of the list d is assumed to be clear from the context. A *state* is an assignment of a value to each variable, i.e. a (total) function from *Var* to D. We assume that there are no undefined values. The set of all states is called the *state space* and is denoted Σ. A typical element of Σ is denoted σ.

A (*semantic*) *substitution* in Σ is defined in the following way. The state σ with c (semantically) substituted for u, denoted $\sigma[c/u]$, is the state which differs from σ only in that it assigns the value c to the variable u. Substitutions of lists of values for lists of distinct variables are defined similarly.

4.3. *The Lattice of Predicates*

By a *predicate* we mean an assignment of a truth value to every state in the state space Σ, i.e. a (total) function from Σ to *Bool*. We denote the set of all predicates *Pred*; typical predicates are P and Q. Instead of $P(\sigma)=tt$ we often write just $P(\sigma)$, saying that *P holds in the state σ.*

Note that every function from Σ to *Bool* is considered a predicate. Thus we do not assume that a predicate can be represented by a finite expression in some language.

By the results of Sect. 2, *Pred* is a complete boolean lattice for the pointwise extended partial order from *Bool*:

$$P \leqq Q \overset{\text{def}}{=} \forall \sigma \cdot (P(\sigma) \leqq Q(\sigma)).$$

Note that we can interpret $P \leqq Q$ as "P implies Q". Also, the meet operation can be interpreted as logical conjunction and the join operation as logical disjunction of predicates (this is a straightforward consequence of the fact that the partial order on *Pred* is the pointwise extension of the implication ordering on *Bool*).

Note that we allow infinite conjunctions (meets) and disjunctions (joins) of predicates. If $\mathscr{P}$ is a set of predicates we have

$$\begin{array}{lll} (\bigwedge \mathscr{P})(\sigma)=tt & \text{iff } P(\sigma)=tt & \text{for all } P \in \mathscr{P}, \\ (\bigvee \mathscr{P})(\sigma)=tt & \text{iff } P(\sigma)=tt & \text{for some } P \in \mathscr{P} \end{array}$$

for all states σ.

Special predicates and notation. If u is a variable and c a value, the *testing predicate* $(u=c)$ is defined by

$$(u=c)(\sigma)=tt \quad \text{iff } \sigma(u)=c. \tag{3}$$

The predicate $(v=d)$ where v and d are lists is defined analogously.

The bottom element of *Pred* is the predicate *false* which assigns the value *ff* to every state and, correspondingly, the top element is the predicate *true* which assigns the value *tt* to every state. The inverse of a predicate P is $\neg P$ ("not P") which assigns *tt* to a state σ if $P(\sigma)=ff$ and vice versa.

As noted above, $P \leqq Q$ means "P implies Q". It would thus be tempting to write $P \Rightarrow Q$ instead of $P \leqq Q$. We will, however, reserve the notation $P \Rightarrow Q$ for the *predicate* defined by

$$(P \Rightarrow Q)(\sigma) \overset{\text{def}}{=} P(\sigma) \leqq Q(\sigma).$$

The partial order on *Pred* can then be characterized in the following way:

$$P \leqq Q \quad \text{if and only if } (P \Rightarrow Q) = \mathit{true}. \tag{4}$$

Substitutions in Σ are extended to *Pred* in a natural way. The predicate $P[d/v]$ assigns to any state σ the same truth value as P assigns to $\sigma[d/v]$, i.e. $P[d/v](\sigma) = P(\sigma[d/v])$. Semantic substitution distributes over meets and joins as well as over negation and implication of predicates (this is a straightforward consequence of the definitions).

We say that a *predicate P depends on the variable u* if there exist values c_1 and c_2 such that $P[c_1/u] \neq P[c_2/u]$. The set of variables that the predicate P depends on is denoted $\mathit{Var}(P)$. We note that if v is a list of variables with $\mathit{Var}(P) \subseteq v$ then for all lists of values d, the predicate $P[d/v]$ is constant (equal to *true* or *false*).

Qualified meets and joins of predicates. Assume that v is a list of distinct variables and that $\{Q_d\}$ is a set of predicates indexed by the set of all lists of values from D of the same length as v. Further assume that P is a predicate with $\mathit{Var}(P) \subseteq v$. We will often take meets and joins over all those predicates Q_d where d ranges over all the lists that make the predicate $P[d/v]$ equal to *true* (we call these *qualified* meets and joins). We introduce the following notations

$$\bigwedge_{d:\,P[d/v]} Q_d \overset{\text{def}}{=} \bigwedge(d\colon P[d/v] = \mathit{true}\colon Q_d), \tag{5}$$

$$\bigvee_{d:\,P[d/v]} Q_d \overset{\text{def}}{=} \bigvee(d\colon P[d/v] = \mathit{true}\colon Q_d). \tag{6}$$

When the variables v can be inferred from the context we sometimes write $\bigwedge_{d:P} Q_d$ instead of $\bigwedge_{d:P[d/v]} Q_d$, and $\bigvee_{d:P} Q_d$ instead of $\bigvee_{d:P[d/v]} Q_d$. Note that the meet $\bigwedge_{d} Q_d$ where d ranges over all lists of the same length as v, is the same as $\bigwedge_{d:\,\mathit{true}} Q_d$. Dually, $\bigvee_{d} Q_d$ is the same as $\bigvee_{d:\,\mathit{true}} Q_d$.

Using the above notations we define *quantified predicates.* If P is a predicate and v a list of variables then we define the predicates $\forall v \cdot P$ and $\exists v \cdot P$ in the following way.

$$\forall v \cdot P \overset{\text{def}}{=} \bigwedge_d P[d/v],$$

$$\exists v \cdot P \overset{\text{def}}{=} \bigvee_d P[d/v].$$

We now prove two lemmas on predicates, showing how any predicate P can be written as a join of simpler predicates.

Lemma 1. *Assume that P is a predicate and that v is a list of variables such that $Var(P) \subseteq v$. Then*

$$\bigvee_{d:P} (v=d) = P.$$

Proof. Let σ be any state in Σ.

$(\bigvee_{d:P} (v=d))(\sigma) = tt$

$\Leftrightarrow$ [definition of qualified join]

$(\bigvee (d: P[d/v] = true: (v=d)))(\sigma) = tt$

$\Leftrightarrow$ [definition of $\vee$ in *Pred*]

$\bigvee (d: P[d/v] = true: (v=d)(\sigma)) = tt$

$\Leftrightarrow$ [$\vee$ in *Bool* is logical disjunction]

$(P[d/v] = true) \wedge ((v=d)(\sigma) = tt)$ for some d

$\Leftrightarrow$ [definition of $(v=d)$]

$(P[d/v] = true) \wedge (\sigma(v) = d)$ for some d

$\Leftrightarrow$ [one point rule]

$P[\sigma(v)/v] = true$

$\Leftrightarrow$ [$Var(P) \subseteq v$ so $P[d/v]$ is constant]

$P(\sigma) = tt.$ □

Lemma 2. *Assume that P is a predicate and that v is any list of variables. Then*

$$\bigvee_d ((v=d) \wedge P[d/v]) = P.$$

Proof. The proof is similar to that of Lemma 1 (see the Appendix). □

4.4. The Lattice of Predicate Transformers

We now introduce *predicate transformers* as (total) functions from predicates to predicates. The set of all predicate transformers, i.e. all total functions from *Pred* to *Pred* is denoted *Ptran*, with typical element S. The partial order on *Ptran* is the pointwise extension of the partial order on *Pred*:

$$S \leqq S' \overset{\text{def}}{=} \forall P \cdot (S(P) \leqq S'(P))$$

This ordering is equivalent to the *refinement ordering* of Back [1, 2]. It is also used by Morris [20] and Gardiner and Morgan [11]. By the results of Sect. 2, *Ptran* is a complete boolean lattice. The bottom element, which we will call **abort**, maps every predicate to the predicate *false* and the top element, called **magic**, maps every predicate to the predicate *true*. Note that $\mathbf{abort} = \bigvee \emptyset$ and $\mathbf{magic} = \bigwedge \emptyset$ and that **abort** and **magic** are the unit elements of the join and meet operators, respectively.

Predicate transformers interpreted as statements. As noted in the introduction, we identify statements with their weakest precondition predicate transformers. The intuitive meaning of the statement S, identified with the predicates transformer S, is the following [9]: S is guaranteed to terminate in a state in which P holds if and only if it is executed in an initial state in which the predicate $S(P)$ holds (where P is an arbitrary predicate). Thus, for example, a statement (predicate transformer) that maps *true* to *true* is guaranteed to terminate for all initial states. Since the statement **abort** maps *true* to *false*, it is never guaranteed to terminate. The statement **magic**, on the other hand, maps *false* to *true*. Since the predicate *false* does not hold in any state, this means that **magic** achieves the impossible; we say it is *miraculous*. Miraculous statements cannot be implemented, but they can be useful in program development [18, 19]. In this respect they resemble imaginary numbers.

Properties of predicate transformers. Four properties of predicate transformers are of special interest to us: strictness, monotonicity, conjunctivity and disjunctivity (we do not consider continuity in this paper, we assume unbounded nondeterminism is always permitted). Assume that S is a predicate transformer. We then have the following definitions:

1. S is *strict* if $S(\mathit{false}) = \mathit{false}$.
2. S is *monotonic* if for any predicates P and Q we have $P \leqq Q \Rightarrow S(P) \leqq S(Q)$.
3. S is *conjunctive* if for any predicates P and Q we have $S(P \wedge Q) = S(P) \wedge S(Q)$.
4. S is *disjunctive* if for any predicates P and Q we have $S(P \vee Q) = S(P) \vee S(Q)$.

Sequential composition of predicate transformers. We denote the functional composition (which we will call *sequential composition*) of two predicate transformers S_1 and S_2 in *Ptran* by $S_1; S_2$. Thus,

$$(S_1; S_2)(P) \overset{\text{def}}{=} S_1(S_2(P)).$$

Obviously *Ptran* is closed with respect to sequential composition. Furthermore, sequential composition preserves both strictness, monotonicity, conjunctivity and disjunctivity. The unit element of sequential composition is the identity predicate transformer **skip**, which maps every predicate to itself.

4.5. The Lattice of Monotonic Predicate Transformers

According to the results in Sect. 2, the *monotonic predicate transformers* form a complete, distributive (but not boolean) sublattice of *Ptran*. We denote this sublattice *Mtran*.

By the results of Sect. 2, meets and joins in *Mtran* yield the same results as in *Ptran*. In particular, *Mtran* has the same bottom and top elements, **abort** and **magic**, as *Ptran*. It is straightforward to show that *Mtran* is closed with respect to sequential composition. Also it should be noted that monotonic predicate transformers need be neither strict, conjunctive nor disjunctive. If D contains only one element, then *Mtran* contains only the three predicate transformers **abort**, **skip** and **magic**. To avoid this trivial case, we assume from now on that *D contains at least two elements.*

Strict and monotonic predicate transformers. By the results of Sect. 2, the predicate transformers that are both monotonic and strict form a distributive sublattice of *Mtran*. We call it *SMtran*. We note that *SMtran* is not strictly speaking a complete sublattice of *Mtran* since $\bigwedge\emptyset$ yields different results in *SMtran* and *Mtran*. However, we have the following result.

Lemma 3. *SMtran is a complete lattice and all meets and joins, except* $\bigwedge\emptyset$, *yield the same result when taken in SMtran as when taken in Mtran. In particular the identities* (MID) *and* (JID) *hold in SMtran.*

5. A Lattice-Based Specification Language

In this section we construct a very general specification language. We refer to this language as *the base language* and denote it $\mathscr{C}^{\perp}$. The commands of our language form a lattice of predicate transformers (the superscript $\perp$ in the name of the language indicates strictness with respect to the bottom predicate *false*).

5.1. Primitive Commands

As a basis for our command lattice we introduce two primitive commands. The first primitive command is the (*multiple*) *substitution command*, $\langle d/v\rangle$ defined by

$$\langle d/v\rangle(Q) \overset{\text{def}}{=} Q[d/v] \tag{7}$$

for all predicates Q, i.e. the command $\langle d/v\rangle$ maps a predicate Q to the predicate $Q[d/v]$. Here v is a list of distinct variables and d is a list of values of the same length as v. Note that if v is finite then $\langle d/v\rangle$ can be written as a sequential composition of its component substitutions. However, if v is infinite, this is not possible.

Our second primitive command is the *(single) test command* which tests for equality. Testing a variable u for equality with a value c is denoted $\langle u=c\rangle$ and is defined by

$$\langle u=c\rangle (Q) \overset{\text{def}}{=} (u=c) \wedge Q \tag{8}$$

for all predicates Q. We also define the multiple test $\langle v=d\rangle$ as the meet of the component tests, thus

$$\langle v=d\rangle (Q) = (v=d) \wedge Q. \tag{9}$$

5.2. *A Lattice of Commands*

Starting from the primitive commands we now generate new commands using the following constructors: sequential composition, meet and join. These constructors have their usual meaning in the lattice of predicate transformers. Thus the commands of $\mathscr{C}^{\perp}$ are defined by

$S ::= \langle d/v\rangle$	(*substitution command*)
$\mid \quad \langle u=c\rangle$	(*test command*)
$\mid \quad S_1; S_2$	(*sequential composition*)
$\mid \quad \bigwedge_{i\in I} S_i$	(*meet*)
$\mid \quad \bigvee_{i\in I} S_i.$	(*join*)

where S, S' and S_i are commands and I is an arbitrary (possibly empty, possibly infinite) index set. We follow the convention that the constructor ";" binds tighter than " $\wedge$ ", which in turn binds tighter than " $\vee$ ".

We note that the meet and join constructors distribute over each other. Furthermore, sequential composition distributes to the right over meets and joins; we have

$$(S_1 \wedge S_2); S = S_1; S \wedge S_2; S,$$
$$(S_1 \vee S_2); S = S_1; S \vee S_2; S$$

for all commands S_1, S_2 and S.

The primitive commands $\langle d/v\rangle$ and $\langle u=c\rangle$ are strict and monotonic. Furthermore, since *SMtran* is closed under sequential composition and it is a complete lattice we cannot generate commands outside *SMtran*. Thus all our commands are in *SMtran*. Also, since our definition states that arbitrary meets and joins are allowed, the set of commands is a complete lattice.

Intuitive interpretation of the commands. The substitution command $\langle d/v \rangle$ is interpretated as assigning the values d to the variables v, leaving the rest of the state unchanged. It always terminates. The test command $\langle u=c \rangle$ terminates without changing the state if the value of the variable u in the initial state is c, otherwise it aborts.

Sequential composition has its usual interpretation. Thus, execution of $S_1; S_2$ in initial state σ_0 is interpretated as first executing S_1 in initial state σ_0 and then, provided that S_1 terminates in some state σ_1, executing S_2 in initial state σ_1.

The base language permits both *demonic* and *angelic* nondeterminism. The meet $S_1 \wedge S_2$ is interpreted as a demonic choice between the two commands S_1 and S_2. Thus, for $S_1 \wedge S_2$ to establish some condition P we require that both S_1 and S_2 establish P. The join $S_1 \vee S_2$, on the other hand, is interpreted as an angelic choice between S_1 and S_2. For $S_1 \vee S_2$ to establish some condition P it is enough that either S_1 or S_2 establishes P.

5.3. Example Commands

As examples, we construct a few commands that will be used later on. We first define *qualified meets and joins* of commands, in the same way as qualified meets and joins of predicates (see (5) and (6)). Thus, if v is a list of distinct variables and $\{S_d\}$ is a set of commands in $\mathscr{C}^{\perp}$, indexed by the set of all lists d of values of the same length as v and P is a predicate with $Var(P) \subseteq v$, then

$$\bigwedge_{d:P} S_d \overset{\text{def}}{=} \bigwedge(d: P[d/v] = true: S_d), \tag{10}$$

$$\bigvee_{d:P} S_d \overset{\text{def}}{=} \bigvee(d: P[d/v] = true: S_d). \tag{11}$$

We now look at a few example commands that will be used as building blocks later on:

Lemma 4. *Assume that P is a predicate and that v is a list of variables such that $Var(P) \subseteq v$. Then*

(a) $(\bigvee_{d:P} \langle v=d \rangle)(Q) = P \wedge Q$,

(b) $(\bigwedge_{d:P} \langle d/v \rangle)(Q) = \forall v \cdot (P \Rightarrow Q)$, *if* $P \neq false$,

(c) $(\bigvee_{d:P} \langle d/v \rangle)(Q) = \exists v \cdot (P \wedge Q)$

for all predicates Q.

If $P = false$ then (b) of Lemma 4 degenerates to $\bigwedge \emptyset$, which is the top element of $\mathscr{C}^{\perp}$ (see (12) below).

Now let v be any list of variables. As special cases of Lemma 4, with $P = true$, we have the following lemma:

Lemma 5. *If v is a list of distinct variables then the following identities hold (d is assumed to range over all lists of values with the same length as v):*

(a) $(\bigvee_d \langle v=d\rangle)(Q)=Q$,

(b) $(\bigwedge_d \langle d/v\rangle)(Q)=\forall v\cdot Q$,

(c) $(\bigvee_d \langle d/v\rangle)(Q)=\exists v\cdot Q$

for all predicates Q.

Proof. Since $Var(true)=\emptyset$, $Var(true)\subseteq v$ holds for all variable lists v. Thus we can use the results of Lemma 4. $\square$

Now we can characterize a few special commands. As stated earlier, V is the list of all variables in Var. By Lemma 5, we have

$$(\bigvee_d \langle d/V\rangle)(Q)=\exists V\cdot Q=\begin{cases} true & \text{if } Q\neq false\\ false & \text{if } Q=false\end{cases} \tag{12}$$

and

$$(\bigwedge_d \langle d/V\rangle)(Q)=\forall V\cdot Q=\begin{cases} true & \text{if } Q=true\\ false & \text{if } Q\neq true.\end{cases} \tag{13}$$

We give the command $\bigvee_d \langle d/V\rangle$ in (12) the name **serve** and the command $\bigwedge_d \langle d/V\rangle$ in (13) the name **avoid**. Note that **serve** is the top element of $SMtran$ (and of $\mathscr{C}^{\perp}$). It is an extremely angelic command which succeeds always when it is possible. On the other hand, **avoid** is an extremely demonic command which is never guaranteed to produce the required final state (however, it always terminates).

We further note that by Lemma 5, the identity predicate transformer **skip** is $\bigvee_d \langle v=d\rangle$. Noting that **abort** is obviously the bottom element of $\mathscr{C}^{\perp}$, we summarize:

$$\textbf{serve}=\bigvee_d \langle d/V\rangle \qquad \textbf{skip}=\bigvee_d \langle v=d\rangle,$$
$$\textbf{avoid}=\bigwedge_d \langle d/V\rangle \qquad \textbf{abort}=\bigvee\emptyset.$$

The command **serve** can also be constructed as $\bigwedge\emptyset$ and **abort** also as $\bigwedge_d \langle v=d\rangle$.

5.4. Properties of the Base Language

The classical healthiness axioms state that the weakest precondition predicate transformers are strict (i.e. satisfy the Law of Excluded Miracle), monotonic and conjunctive. Furthermore, the conjugated basic statement introduced in [5] is disjunctive. We now examine our command lattice with respect to these properties.

Strictness and monotonicity. As noted in Sect. 5.2, all commands of the base language are both strict and monotonic. In particular, the predicate transformer **magic** cannot be defined in this language.

Conjunctivity. The base language commands are not necessarily conjunctive. A counterexample is the command $\langle d/v\rangle \vee \langle d'/v\rangle$ where d and d' are distinct. Applying this command to the predicate $(v=d)\wedge(v=d')$ we have

$(\langle d/v\rangle \vee \langle d'/v\rangle\, ((v=d)\wedge(v=d'))$

$=$ [no state can assign both d and d' to v]

$(\langle d/v\rangle \vee \langle d'/v\rangle)\,(\mathit{false})$

$=$ [strictness]

false

but, on the other hand,

$(\langle d/v\rangle \vee \langle d'/v\rangle)\,(v=d)\wedge(\langle d/v\rangle \vee \langle d'/v\rangle)\,(v=d')$

$=$ [$\vee$ is defined by pointwise extension]

$(\langle d/v\rangle\,(v=d)\vee\langle d'/v\rangle\,(v=d))\wedge(\langle d/v\rangle\,(v=d')\vee\langle d'/v\rangle\,(v=d'))$

$=$ [definition of $\langle d/v\rangle$]

$((d=d)\vee(d'=d))\wedge((d=d')\vee(d'=d'))$

$=$ [d and d' are distinct]

$(\mathit{true}\vee\mathit{false})\wedge(\mathit{false}\vee\mathit{true})$

$=$ [*true* and *false* are top and bottom elements of *Pred*]

true.

Since the primitive commands are conjunctive the above counterexample shows that joins do not generally preserve conjunctivity.

Disjunctivity. The counterexample to disjunctivity is the dual of the counterexample to conjunctivity, i.e. the command $\langle d/v\rangle\wedge(d'/v)$ applied to the predicate $(v=d)\vee(v=d')$. Thus meets generally do not preserve disjunctivity.

We summarize the above results in the following theorem.

Theorem 2. *All commands of $\mathscr{C}^{\perp}$ are strict and monotonic. The primitive commands are conjunctive and disjunctive. Sequential composition and arbitrary meets preserve conjunctivity while sequential composition and arbitrary joins preserve disjunctivity.*

5.5. Monotonicity in Subcommands

Let $T[X]$ be an expression formed using the symbol X in addition to the primitive commands and the constructors of $\mathscr{C}^{\perp}$. This means that for any command S, $T[S]$ is also a command. We say that T is a *command with a command variable* and that S is a *subcommand* of $T[S]$. An important property is *monotonicity with respect to subcommand replacement.*

Theorem 3. *Assume that T is a command with a command variable. Then T is monotonic with respect to subcommand replacement,* i.e.

$$\forall S, S' \cdot (S \leqq S' \Rightarrow T[S] \leqq T[S']).$$

Proof. Assume that S_1, S_1', S_2 and S_2' are commands of $\mathscr{C}^{\perp}$ with $S_1 \leqq S_1'$ and $S_2 \leqq S_2'$. It is then straightforward to prove that

$$S_1 ; S_2 \leqq S_1' ; S_2',$$
$$S_1 \wedge S_2 \leqq S_1' \wedge S_2' \quad \text{and}$$
$$S_1 \vee S_2 \leqq S_1' \vee S_2'.$$

Using structural induction we can now prove the theorem. □

6. A Dual Language with Miracles

The base language $\mathscr{C}^{\perp}$ does not permit miraculous (nonstrict) commands. In this section we consider a command lattice $\mathscr{C}^{\perp}$, which is dual to $\mathscr{C}^{\perp}$ and which contains miraculous commands (the superscript $\top$ denoting strictness with respect to the top predicate *true*). We begin by defining the dual of any predicate transformer and prove some properties of duals.

6.1. Definition of the Dual Language

For every predicate transformers S we define its *dual* S^0 by

$$S^0(Q) \overset{\text{def}}{=} \neg S(\neg Q) \tag{14}$$

for all predicates Q.

The following lemma shows the relations between a predicate transformer and its dual.

Lemma 6. *Let S and S_i be arbitrary predicate transformers in Ptran (where I is an arbitrary index set). Then*

(a) $(S^0)^0 = S$
(b) S *monotonic* $\Leftrightarrow S^0$ *monotonic*
(c) $(S_1 ; S_2)^0 = S_1^0 ; S_2^0$
(d) $S_1 \leqq S_2 \Leftrightarrow S_2^0 \leqq S_1^0$
(e) $(\bigwedge S_i)^0 = \bigvee S_i^0$
(f) $(\bigvee S_i)^0 = \bigwedge S_i^0$
(g) S *is conjunctive* $\Leftrightarrow S^0$ *is disjunctive*

(h) *S is disjunctive* $\Leftrightarrow S^0$ *is conjunctive.*

We now define the *dual base language* as $\mathscr{C}^\top = \{S^0 \mid S \in \mathscr{C}^\perp\}$. The following theorem shows the duality between $\mathscr{C}^\perp$ and $\mathscr{C}^\top$.

Theorem 4. *The dual language $\mathscr{C}^\top$ is a complete lattice and $(\mathscr{C}^\top, \geqq)$ is isomorphic with $(\mathscr{C}^\perp, \leqq)$. The commands of $\mathscr{C}^\top$ are monotonic and strict with respect to the predicate true* (i.e. *they map true to true*).

Proof. Let the function ϕ be defined by $\phi(S) = S^0$ for all $S \in Ptran$. Part (a) of Lemma 6 shows that ϕ is bijective. Furthermore, (e) and (f) show that ϕ is a lattice-isomorphism between $(Ptran, \leqq)$ and $(Ptran, \geqq)$. Since $\mathscr{C}^\top$ is defined as $\phi(\mathscr{C}^\perp)$, it follows from the above that $(\mathscr{C}^\top, \geqq)$ is isomorphic with $(\mathscr{C}^\perp, \leqq)$. Thus, $\mathscr{C}^\top$ is a complete lattice.

Lemma 6(b) shows that all predicate transformers of $\mathscr{C}^\top$ are monotonic. Finally, we show that every command of $\mathscr{C}^\top$ is strict with respect to *true*. Assume that S is a command in $\mathscr{C}^\perp$. Then from the definition of S^0 and the fact that S is strict, it follows that

$$S^0(true) = \neg S(false) = \neg false = true$$

so S^0 is strict with respect to *true*. □

A command which maps *true* to *true* is interpreted as always terminating. Thus the commands of $\mathscr{C}^\top$ never abort. However, since they are not necessarily strict, they may be miraculous (see Sect. 4.4).

6.2. Generating the Dual Language

Because of the duality shown in Theorem 4 and Lemma 6, we can generate $\mathscr{C}^\top$ using the constructors ";", " $\wedge$ " and " $\vee$ " and the duals of the primitive commands of $\mathscr{C}^\perp$. The substitution command is its own dual, while the dual of the test command is the *miraculous test command* $\langle u = c\rangle^0$, with

$$\langle u = c\rangle^0 (Q) = (u = c) \Rightarrow Q. \tag{15}$$

It is miraculous as

$$\langle u = c\rangle^0 (false) = ((u = c) \Rightarrow false) = \neg (u = c)$$

and D contains at least one value different from c. The top element of $\mathscr{C}^\top$ is **magic** and the bottom element is **avoid**. Note that their constructions are dual to the constructions of the bottom and top elements of $\mathscr{C}^\perp$;

$$\begin{aligned} &\mathbf{abort} = \bigvee \emptyset \quad (\text{in } \mathscr{C}^\perp) &\quad& \mathbf{magic} = \bigwedge \emptyset \quad (\text{in } \mathscr{C}^\top) \\ &\mathbf{serve} = \bigvee_d \langle d/V\rangle && \mathbf{avoid} = \bigwedge_d \langle d/V\rangle \end{aligned}$$

Since the cardinality of the value set D is at least 2, we can construct **abort** in $\mathscr{C}^{\perp}$ and **magic** in $\mathscr{C}^{\top}$ without using the somewhat ambiguous empty meets and joins, in a way which emphasizes their duality:

$$\mathbf{abort} = \bigwedge_{d} \langle v = d \rangle \qquad \mathbf{magic} = \bigvee_{d} \langle v = d \rangle^{0}$$

where v is an arbitrary nonempty list of distinct variables.

6.3. *Combining the Two Languages*

Since the command $\langle u = c \rangle^{0}$ is miraculous, it cannot be generated within $\mathscr{C}^{\perp}$. Conversely, the test command $(u = c)$ cannot be generated within $\mathscr{C}^{\top}$, since it is possibly nonterminating.

Thus, if we construct commands using sequential composition, meets and joins starting from the three primitive statements $\langle d/v \rangle$, $\langle u = c \rangle$ and $\langle u = c \rangle^{0}$ we get an *extended base language*, which we denote $\mathscr{C}$. This is a command lattice which contains both strict and miraculous predicate transformers. In fact, as we will show in the next section, it contains all monotonic predicate transformers.

There are other ways of generating the language $\mathscr{C}$. Adding the single command **magic** to $\mathscr{C}^{\perp}$ (or, dually, adding **abort** to $\mathscr{C}^{\top}$) is sufficient to get all the commands in $\mathscr{C}$. This is seen from the following lemma.

Lemma 7. *Let u be an arbitrary variable and c an arbitrary value. Then*

$$\langle u = c \rangle^{0} = \bigvee_{c' \neq c} ((u = c'); \mathbf{magic}) \vee \bigvee_{c'} (u = c') \tag{16}$$

$$\langle u = c \rangle = \bigwedge_{c' \neq c} (\langle u = c' \rangle^{0}; \mathbf{abort}) \wedge \bigwedge_{c'} \langle u = c' \rangle^{0}. \tag{17}$$

Note that the second disjunct on the right hand side of (16) is $\bigvee_{c'} (u = c') = \mathbf{skip}$ while the second conjunct on the right hand side of (17) is $\bigwedge_{c'} \langle u = c \rangle^{0} = \mathbf{skip}$ (so **skip** is its own dual).

7. Completeness of the Command Languages

In the preceding sections we defined command languages that are lattices of predicate transformers. We showed that $\mathscr{C}^{\perp}$ is a complete sublattice of *SMtran*, the lattice of strict monotonic predicate transformers. Similarly, one can show that $\mathscr{C}$ is a complete sublattice of *Mtran*, the lattice of monotonic predicate transformers. In this section we generalize these results. We show that in fact all strict monotonic predicate transformers can be generated within $\mathscr{C}^{\perp}$, i.e. that $\mathscr{C}^{\perp} = \mathit{SMtran}$. In order to show this, we first construct a base for the lattice *SMtran* and then show how every element of this base can be generated in

$\mathscr{C}^{\perp}$. Similarly, we construct a base for *Mtran* and show that every element of this base can be generated in $\mathscr{C}$.

We first define the concept of base for a complete lattice. A subset B of a complete lattice L is a $\vee$*-base* (or just *base*) for L if every element of L can be written as a join of elements in B (note that the bottom element of a complete lattice can be written as the empty join). Equivalently, B is a base if $x = \bigvee(b \in B : b \leqq x : b)$ for all x in L.

7.1. Bases for Function Lattices

We now give a construction of a base for the function lattice $[L \to L]_M$, given a base for L. This base can be considered to be a pointwise extension of the base for L.

Lemma 8. *Assume that L is a complete lattice and that B is a base for L.*
For every $x \in K$ and every $b \in B$ define $g_{x,b}$ by

$$g_{x,b}(y) = \begin{cases} b & \textit{if } x \leqq y \\ \perp_L & \textit{otherwise.} \end{cases}$$

Then any monotonic function f in $[L \to L]$ can be written as

$$f = \bigvee_{x \in L} (\bigvee_{b \leqq f(x)} g_{x,b})$$

and $\{g_{x,b} \mid x \in K, b \in B\}$ is a base for $[L \to L]_M$.

Proof. From the definition of $g_{x,b}$ it follows that every $g_{x,b}$ is monotonic, so every $g_{x,b}$ is in $[L \to L]_M$.

Now let f be an arbitrary monotonic function on L and define f_x by

$$f_x = \bigvee_{b \leqq f(x)} g_{x,b}.$$

Then for an arbitrary $y \in L$ we have

$$f_x(y) = \bigvee_{b \leqq f(x)} g_{x,b}(y) = \begin{cases} f(x) & \text{if } x \leqq y \\ \perp_L & \text{otherwise.} \end{cases}$$

This means that for an arbitrary $y \in L$,

$$(\bigvee_{x \in K} (\bigvee_{b \leqq f(x)} g_{x,b}))(y)$$

$= [$definition of $f_x]$

$$\bigvee_{x \in K} (f_x(y))$$

$= [f_x(y)$ is $\perp$ unless $x \leqq y]$

$$\bigvee_{x \leqq y} (f_x(y))$$

$$= [f_x(y) = f(x) \text{ when } x \leqq y]$$

$$\bigvee_{x \leqq y} (f(x))$$

$$= [f \text{ is monotonic}]$$

$$f(y)$$

which completes the proof. □

The same technique is used to construct a base for the strict monotonic functions on the lattice L, given a base for L.

Lemma 9. *Assume L and B as in the preceding lemma. For every $x \in L$ except $x = \bot$ and every $b \in B$ define $g_{x,b}$ by*

$$g_{x,b}(y) = \begin{cases} b & \textit{if } x \leqq y \\ \bot_L & \textit{otherwise.} \end{cases}$$

Then any strict monotonic function f in $[L \to L]$ can be written as

$$f = \bigvee_{x \in L} (\bigvee_{b \leqq f(x)} g_{x,b})$$

and $\{g_{x,b} \mid \bot_L \neq x \in K, i \in I\}$ is a base for $[L \to L]_{SM}$.

Proof. The proof proceeds exactly as the proof of Lemma 8. □

A base for Mtran. We now construct a base for the lattice of monotonic predicate transformers *Mtran*. We begin by observing that the set $\{tt\}$ is a base for *Bool*. By a proof, similar to the proof of Lemma 8, one can show that the one-state predicates $\{b_\sigma \mid \sigma \in \Sigma\}$ form a base for *Pred*, where b_σ is defined by

$$b_\sigma(\sigma') = \begin{cases} tt & \text{if } \sigma' = \sigma \\ ff & \text{otherwise.} \end{cases}$$

Using this base, any predicate P can be written as

$$P = \bigvee_{\sigma : P(\sigma)} b_\sigma .$$

Using Lemma 8 we construct a base for *Mtran*. This base is the set of predicate transformers $\{G_{P,\sigma} \mid P \in Pred, \sigma \in \Sigma\}$ where $G_{P,\sigma}$ is defined by

$$G_{P,\sigma}(Q) = \begin{cases} b_\sigma & \text{if } P \leqq Q \\ \mathit{false} & \text{otherwise} \end{cases} \tag{18}$$

for all states σ and all predicates P and Q.

The base element $G_{P,\sigma}$ can be characterized in the following way: $G_{P,\sigma}$ is a command which establishes the condition P (and, by monotonicity, all conditions implied by P) when executed in the initial state σ, and aborts for all other initial states.

A base for SMtran. We now construct a base for the lattice of strict monotonic predicate transformers *SMtran* in the same way as we constructed a base for *Mtran*. By Lemma 9, a base is the set of predicate transformers $\{G_{P,\sigma} \mid false \neq P \in Pred, \sigma \in \Sigma\}$ where $G_{P,\sigma}$ is defined by (18).

7.2. *Completeness of $\mathscr{C}^{\perp}$*

The following lemma shows how every element $G_{P,\sigma}$ of the base for *SMtran* can be generated within $\mathscr{C}^{\perp}$.

Lemma 10. *Let V be the list of all variables in Var, P an arbitrary predicate with $P \neq false$ and σ an arbitrary state. Then*

$$G_{P,\sigma} = \langle V = \sigma(V) \rangle ; \bigwedge_{d:P} \langle d/V \rangle$$

Proof. We first note that for arbitrary predicates P and Q,

$$(\bigwedge_{d:P} \langle d/V \rangle)(Q) = \begin{cases} true & \text{if } P \leqq Q \\ false & \text{otherwise.} \end{cases} \tag{19}$$

This is seen as follows:

$$\begin{aligned} &(\bigwedge_{d:P} \langle d/V \rangle)(Q) \\ &= [\text{Lemma 4(b)}] \\ &\forall V \cdot (P \Rightarrow Q) \\ &= [\text{definition of quantified predicate, (4)}] \\ &\begin{cases} true & \text{if } P \leqq Q \\ false & \text{otherwise.} \end{cases} \end{aligned}$$

We further note that if σ is an arbitrary state in Σ, then

$$\langle V = \sigma(V) \rangle (true) = (V = \sigma(V)) = b_{\sigma}. \tag{20}$$

Now we move to the actual proof:

$$\begin{aligned} &(\langle V = \sigma(V) \rangle ; (\bigwedge_{d:P} \langle d/V \rangle))(Q) \\ &= [\text{definition of sequential composition}] \\ &\langle V = \sigma(V) \rangle ((\bigwedge_{d:P} \langle d/V \rangle)(Q)) \\ &= [(19)] \\ &\begin{cases} \langle V = \sigma(V) \rangle (true) & \text{if } P \leqq Q \\ \langle V = \sigma(V) \rangle (false) & \text{otherwise} \end{cases} \\ &= [(20), \text{strictness}] \end{aligned}$$

$$\begin{cases} b_\sigma & \text{if } P \leqq Q \\ \mathit{false} & \text{otherwise} \end{cases}$$

$=[(18)]$

$G_{P,\sigma}(Q)$. □

We can now state the completeness theorem.

Theorem 5. *Every strict monotonic predicate transformer can be generated within the base language $\mathscr{C}^\perp$. Or, equivalently,*

$$\mathscr{C}^\perp = SMtran.$$

Proof. Lemma 10 shows how to generate any element $G_{P,\sigma}$ in a base for *SMtran*. Since every element in *SMtran* can be written as a join of elements in the base, we have shown that every element in *SMtran* can be constructed in $\mathscr{C}^\perp$.

In fact, by Lemma 9 any element S of *SMtran* can be constructed as

$$S = \bigvee_{P \in Pred} \bigvee_{\sigma : S(P)(\sigma)} (\langle V = \sigma(V)\rangle ; \bigwedge_{d:P} \langle d/V\rangle). \quad \square$$

We note that a dual completeness theorem holds for $\mathscr{C}^\top$:

Corollary 1. *Every monotonic predicate transformer that is strict with respect to the predicate true can be generated within the dual base language $\mathscr{C}^\top$.*

Proof. We first note that the predicate transformers

$$\{G^0_{P,\sigma} \mid false \neq P \in Pred, \sigma \in \Sigma\}$$

form a $\wedge$-base for the set of all predicate transformers that are strict with respect to *true*. We then use the dual of the construction in Lemma 10 to generate this $\wedge$-base in $\mathscr{C}^\top$. □

7.3. *Completeness of $\mathscr{C}$*

The completeness of the extended base language $\mathscr{C}$ is shown in the following theorem.

Theorem 6. *Every monotonic predicate transformer can be generated within the extended base language $\mathscr{C}$. Or, equivalently,*

$$\mathscr{C} = Mtran.$$

Proof. The same construction of an arbitrary $G_{P,\sigma}$ as in Lemma 10 can be used in $\mathscr{C}$. Furthermore, this construction holds also when $P = false$, since in this case

$$\langle V = \sigma(V)\rangle ; \bigwedge_{d:P} \langle d/V\rangle = \langle V = \sigma(V)\rangle ; \bigwedge \emptyset = \{b_\sigma\} ; \mathbf{magic} = G_{P,\sigma}.$$

The rest of the proof proceeds exactly as the proof of Theorem 6. □

8. Derivation of a Simple Language in the Command Lattice

The base languages constructed in the previous sections are infinitary, in the sense that they permit meets and joins over arbitrary sets of commands. The definition of commands as predicate transformers is highly semantic, since predicates are considered to be functions from Σ to *Bool*.

In this section we consider a finitary specification language with a syntactic notion of predicates. The language is an extension of the language of guarded commands of Dijkstra [9]. We identify the statements of this language with their weakest precondition predicate transformers. Our aim is to show how this language can be embedded in the base language $\mathscr{C}^{\perp}$.

8.1. *The Specification Language*

We now consider a slightly modified version of the language of Back [5] (subsequently called the *specification language*, denoted *Stat*$^{\perp}$). This language permits input-output-specifications to be embedded as statements in a program. It permits both demonic and angelic nondeterminism, but not miracles. The nondeterminism of a statement can be unbounded. Ordinary program statements form a sublanguage of *Stat*$^{\perp}$. The statements of *Stat*$^{\perp}$ are defined by

$S ::=$	$v := v' \cdot P$	(*demonic assignment statement*)
$\mid$	$\overline{v := v' \cdot P}$	(*angelic assignment statement*)
$\mid$	$[\mathbf{var}\ v; S]$	(*block*)
$\mid$	X	(*statement variable*)
$\mid$	$S_1; S_2$	(*sequential composition*)
$\mid$	$\mathbf{if}\ b_1 \to S_1 \parallel b_2 \to S_2\ \mathbf{fi}$	(*conditional composition*)
$\mid$	$\mu X \cdot T$	(*recursive composition*).

Here v is a list of variables, b_1 and b_2 are predicates and S, S_1 and S_2 are statements. T is an expression containing statements, statement constructors and the statement variable X. In the assignment statements, v' is a list of *auxiliary variables* (i.e. variables not belonging to *Var*), and P is a predicate formula which may refer to the auxiliary variables in v'.

Semantic and syntactic predicates. Since the base language treats predicates as semantic objects and we now consider a language where predicates are syntactic objects we must be careful.

To avoid confusion, we refer to syntactic predicates as *predicate formulas*. Assume that every value d in D has a name d (so that every value can be considered a term). It is then easy to show that every predicate formula P, which refers only to variables in *Var*, has a straightforward meaning as a predicate in *Pred*. Furthermore, this meaning function is preserved by connectives, quantifications and substitutions of values d for variables v.

However, we note that predicate formulas may refer to *auxiliary variables*, not belonging to *Var*. Such formulas do not correspond to predicates in *Pred* unless the auxiliary variables have been removed by substitution or bound by quantification.

Meaning of the primitive statements. As noted above, we identify the statements of $Stat^{\perp}$ with their weakest precondition predicate transformers. Thus every statement is considered to be a function from predicate formulas to predicate formulas.

The effect of the demonic assignment statement is to assign values v' nondeterministically to the variables v, so that the condition P becomes established (unbounded nondeterminism is permitted). Formally,

$$(v := v' \cdot P)(Q) \overset{\text{def}}{=} (\exists v' \cdot P) \wedge \forall v' \cdot (P \Rightarrow Q[v'/v]) \tag{21}$$

for all predicate formulas Q. Note that v' is a list of auxiliary variables, so we can assume that the predicate formula Q does not refer to v'.

The angelic assignment statement is like the corresponding demonic statement but executed with angelic nondeterminism,

$$(\overline{v := v' \cdot P})(Q) \overset{\text{def}}{=} \exists v' \cdot (P \wedge Q[v'/v]). \tag{22}$$

Meaning of constructors. For the block statement, sequential and conditional composition we give the following definitions:

$$([\mathbf{var}\ v; S])(Q) \overset{\text{def}}{=} \forall v \cdot S(\forall v \cdot Q)$$

$$(S_1 ; S_2)(Q) \overset{\text{def}}{=} S_1(S_2(Q))$$

$$(\mathbf{if}\ b_1 \rightarrow S_1 \,\|\, b_2 \rightarrow S_2\ \mathbf{fi})(Q) \overset{\text{def}}{=} (b_1 \vee b_2) \wedge (b_1 \Rightarrow S_1(Q)) \wedge (b_2 \Rightarrow S_2(Q)).$$

The block statement assumes that no redeclaration of variables is permitted. The local variables become undefined (i.e. their value can be considered to be arbitrary) outside the block, as shown by the quantification $\forall v \cdot Q$ in the definition. Thus our definition for the block differs slightly from the corresponding definitions of [2, 18, 19, 20], where it is assumed that no predicates refer to the local variables.

The meaning of the recursive composition $\mu X \cdot T(X)$ is defined in the following way. Assume that $T(X)$ is an expression built up of the primitive statements, the statement variable X and sequential and conditional composition. Then $\lambda X \cdot T(X)$ is a monotonic function on the complete lattice of monotonic predicate transformers. Thus, by the Knaster-Tarski fixpoint theorem, $\lambda X \cdot T(X)$ has a least fixpoint. This least fixpoint is chosen as the meaning of $\mu X \cdot T(X)$. This definition can be used inductively to define the meaning of $\mu X \cdot T(X)$ in the case when the expression $T(X)$ itself contains recursive compositions.

Definitions of recursion using least fixpoints have been used in e.g. [10, 13, 22]. We assume that the logic is sufficiently expressive, so that these least fixpoints can be expressed as predicate formulas, see [3, 13].

The following lemma shows that $Stat^{\perp}$ is in fact a sublanguage of $\mathscr{C}^{\perp}$.

Lemma 11. *All statements in $Stat^{\perp}$ are strict and monotonic.*

Proof. The primitive statements of $Stat^{\perp}$ are easily shown to be strict and monotonic. It is also straightforward to show that sequential and conditional composition preserve strictness and monotonicity. This implies that $T(X)$ in any recursion $\mu X \cdot T(X)$ can be viewed as a monotonic function on the complete lattice *SMtran*. Thus it has a least fixpoint in *SMtran*. Furthermore, since $SMtran = \{S \in Mtran \mid S \leqq \mathbf{serve}\}$, this least fixpoint is also the least fixpoint of $T(X)$, viewed as a function on *Mtran*. Thus $\mu X \cdot T(X)$ is in *SMtran*, i.e. it is strict and monotonic. $\square$

Derived statements. The specification language can be extended with *derived statements*, i.e. statements defined in terms of the other constructs of the language. As an example, we define two derived statements, the *assert statement* and the (ordinary) *multiple assignment statement*. The assert statement is defined as a demonic assignment statement $\varepsilon{:=}\varepsilon \cdot P$ on the empty list ε of variables. We use the notation $\{P\}$ for this statement. Thus the following holds:

$$\{P\}(Q) = P \wedge Q.$$

Recalling the definitions of the statements **skip** and **abort**, we note that they can be written as special case of the assert statement; $\mathbf{skip} = \{true\}$ and $\mathbf{abort} = \{false\}$.

The multiple assignment $v{:=}e$ where e is a list of expressions (i.e. functions from Σ to D) is defined as the demonic assignment $v{:=}v' \cdot (v' = e)$ or the angelic assignment $\overline{v{:=}v' \cdot (v' = e)}$. Thus the following holds:

$$(v{:=}e)(Q) = Q[e/v].$$

A game theoretic interpretation of the specification language. The intuitive interpretation of statements defined as predicate transformers (Sect. 4.4) has to be somewhat extended to cover the angelic assignment statement of our specification language. The non-determinism of the demonic assignment statement is demonic, i.e. $(v{:=}v' \cdot P)(Q)$ holds in a state σ_0 if and only if *all* possible executions of $v{:=}v' \cdot P$ in initial state σ_0 lead to final states where Q holds. On the other hand, the nondeterminism of the angelic assignment statement is angelic, i.e. $(\overline{v{:=}v' \cdot P})(Q)$ holds in a state σ_0 if and only if there *exists* a possible execution of $\overline{v{:=}v' \cdot P}$ in initial state σ_0 which leads to a final state where Q holds. The execution of a statement in the specification language can be interpreted as a game between the system (the demon) and the user (the angel). The demon chooses the values in the demonic assignment statement and the angel chooses the values in the angelic assignment statement. Also, the demon chooses the branch of the **if** statements in the case when both branches are enabled. Assume that the statement S is executed in the initial state σ_0. The angel tries to make the execution terminate in a state where Q holds, whereas the demon tries to prevent this. The state σ_0 belongs to the weakest precondition of S to establish

Q, so $S(Q)$ holds in σ_0, iff the angel has a *winning strategy*, i.e. no matter what choices the demon makes, the angel can force the execution to terminate in a final state where Q holds.

8.2. Construction of the Specification Language Statements in the Base Language

Since every statement in $Stat^{\perp}$ can be seen as a command in $\mathscr{C}^{\perp}$, the completeness theorems of Sect. 7 show that every statement in $Stat^{\perp}$ can be constructed in $\mathscr{C}^{\perp}$. However, the constructions in the completeness theorems are very complex and require total pointwise knowledge of the statement that is to be constructed. We now show how statements of the specification language $Stat^{\perp}$ can be constructed inductively, with the syntactic structure of the statements as a starting point.

Demonic assignment statement. We begin by showing how the demonic assignment statement $v{:=}v'\cdot P$ can be constructed. This will be done in two steps. In the first step we show how to construct the *demonic update statement* $v\cdot P$, which nondeterministically assigns a value to v, such that P holds. This statement is defined by

$$(v\cdot P)(Q)\overset{\text{def}}{=}(\exists v\cdot P)\wedge\forall v\cdot(P\Rightarrow Q). \tag{23}$$

In the second step we make use of the demonic update statement to construct the demonic assignment statement.

Let us first assume that $Var(P)\subseteq v$. Then the statement $v\cdot P$ is constructed in the following way (note how the two conjuncts in the construction correspond to the two conjuncts in (22)).

Lemma 12. *Assume that $Var(P)\subseteq v$. Then*

$$v\cdot P=(\bigvee_{d:P}\langle d/v\rangle);\mathbf{serve}\wedge(\bigwedge_{d:P}\langle d/v\rangle). \tag{24}$$

Proof. If $P=false$ then $\bigvee_{d:P}\langle d/v\rangle=\mathbf{abort}$, so the right hand side, applied to any predicate Q, yields *false*, which $v\cdot P$ also does. Now consider the case when $P\neq false$. We have to show that for any predicate formula Q, the right hand side applied to Q equals $(v\cdot P)(Q)$. If $Q=false$ then this holds by the strictness of all statements in $\mathscr{C}^{\perp}$. Therefore assume that $Q\neq false$. Then

$$((\bigvee_{d:P}\langle d/v\rangle);\mathbf{serve}\wedge(\bigwedge_{d:P}\langle d/v\rangle))(Q)$$

$=$ [definitions of constructors]

$$(\bigvee_{d:P}\langle d/v\rangle)(\mathbf{serve}(Q))\wedge(\bigwedge_{d:P}\langle d/v\rangle)(Q)$$

$=$ [definition of **serve**]

$$(\bigvee_{d:P} \langle d/v \rangle)(true) \wedge (\bigwedge_{d:P} \langle d/v \rangle)(Q)$$

$= $ [Lemma 4(b)]

$$(\bigvee_{d:P} \langle d/v \rangle)(true) \wedge \forall v \cdot (P \Rightarrow Q)$$

$= $ [Lemma 4(c)]

$$\exists v \cdot (P \wedge true) \wedge \forall v \cdot (P \Rightarrow Q)$$

$= $ [$true$ is top element in $Pred$]

$$\exists v \cdot P \wedge \forall v \cdot (P \Rightarrow Q). \quad \square$$

Note that since the statements in $Stat^{\perp}$ are defined only for predicate formulas, the base language construction in (24) should be interpretated as restricted to those predicates that correspond to some predicate formula.

We now consider the general case when $Var(P) \subseteq v$ does not hold. Let w be the list of variables that the predicate formula P refers to and which are not in v. Then for any list of values d of the same length as w, $v \cdot P[d/w]$ is an update statement of the kind considered in Lemma 12.

Lemma 13. *Let v be a list of variables and P an arbitrary predicate formula. Let w be the list of variables that P depends on and which are not in v. Then*

$$v \cdot P = \bigvee_{d} (\langle w = d \rangle ; v \cdot P[d/w])$$

where $v \cdot P[d/w]$ is constructed as in Lemma 12.

Proof. We consider an arbitrary $Q \neq false$.

$$(\bigvee_{d} (\langle w = d \rangle ; v \cdot P[d/w]))(Q)$$

$= $ [definitions of constructors]

$$\bigvee_{d} (\langle w = d \rangle (v \cdot P[d/w](Q)))$$

$= $ [definition of test command]

$$\bigvee_{d} ((w = d) \wedge v \cdot P[d/w](Q))$$

$= $ [Lemma 12]

$$\bigvee_{d} ((w = d) \wedge \exists v \cdot P[d/w] \wedge \forall v \cdot (P[d/w] \Rightarrow Q))$$

$= $ [using the conjunct $(w = d)$]

$$\bigvee_{d} ((w = d) \wedge (\exists v \cdot P[d/w] \wedge \forall v \cdot (P[d/w] \Rightarrow Q)[d/w]))$$

$= $ [substitution distributes over connectives]

$$\bigvee_{d} ((w = d) \wedge (\exists v \cdot P \wedge \forall v \cdot (P \Rightarrow Q))[d/w])$$

$= $ [Lemma 2]

$\exists v \cdot P \wedge \forall v \cdot (P \Rightarrow Q)$. $\square$

In Lemma 13, the disjunction over d is needed to find the value of w in the state.

We can now construct the demonic assignment statement making use of the update statement.

Theorem 7. *Consider the statement $v := v' \cdot P$, where P is a predicate formula. Then*

$$(v := v' \cdot P) = \bigvee_d (\langle v = d \rangle ; v \cdot P[d, v/v, v']).$$

Proof. The formula P may refer to variables from *Var* and possibly to the auxiliary variables in v'. Thus for any list of values d, $P[d, v/v, v']$ refers to only to variables in *Var*, and $v \cdot P[d, v/v, v']$ is an update statement that can be constructed as in Lemma 13.

Now consider an arbitrary predicate $Q \neq false$.

$(\bigvee_d (\langle v = d \rangle ; v \cdot P[d, v/v, v']))(Q)$

$=$ [definition of command constructors)

$\bigvee_d (\langle v = d \rangle (v \cdot P[d, v/v, v'](Q))$

$=$ [definition of $\langle v = d \rangle$, demonic update statement]

$\bigvee_d ((v = d) \wedge \exists v \cdot P[d, v/v, v'] \wedge \forall v \cdot (P[d, v/v, v'] \Rightarrow Q))$

$=$ [renaming bound variables (by definition, Q does not refer to v')]

$\bigvee_d ((v = d) \wedge \exists v' \cdot P[d/v] \wedge \forall v' \cdot (P[d/v] \Rightarrow Q[v'/v]))$

$=$ [substitution distributes over connectives, $Q[v'/v]$ does not refer to v]

$\bigvee_d ((v = d) \wedge (\exists v' \cdot P \wedge \forall v' \cdot (P \Rightarrow Q[v'/v]))[d/v])$

$=$ [Lemma 2]

$\exists v' \cdot P \wedge \forall v' \cdot (P \Rightarrow Q[v'/v])$. $\square$

In this theorem the disjunction over d is again needed to find the correct initial values of v.

Angelic assignment statement. The construction of the angelic assignment statement is similar to that of the corresponding demonic statement. We first show how to construct the *angelic update statement* $\overline{v \cdot P}$ which is the angelic counterpart to $v \cdot P$, defined by

$$\overline{v \cdot P}(Q) = \exists v \cdot (P \wedge Q).$$

This statement is then used in the construction of $\overline{v := v' \cdot P}$.

Lemma 14. *Assume that v is a list of variables and that P is a predicate formula such that $Var(P) \subseteq v$. Then*

$$\overline{v \cdot P} = \bigvee_{d:P} \langle d/v \rangle.$$

Proof. Immediate from Lemma 4(c). $\square$

We now drop the assumption that $Var(P) \subseteq v$. Let w be the list of variables that P refers to and which are not in v. Then for any list of values d of the same length as w, $\overline{v \cdot P[d/w]}$ is an update statement of the kind considered in Lemma 14.

Lemma 15. *Assume that v is a list of variables and that P is an arbitrary predicate formula. Let w be the list of variables that P refers to and which are not in v. Then*

$$\overline{v \cdot P} = \bigvee_{d} (\langle w = d \rangle ; \overline{v \cdot P[d/w]}).$$

Proof. The proof follows the proof of Lemma 13. $\square$

Now we can construct the angelic assignment statement.

Theorem 8. *Consider the statement $\overline{v := v' \cdot P}$, where P is a predicate formula. Then*

$$\overline{v := v' \cdot P} = \bigvee_{d} (\langle v = d \rangle ; \overline{v \cdot P[d, v/v, v']}).$$

Proof. The proof follows the proof of Theorem 7. $\square$

The block statement

Theorem 9. *Assume v is a list of variables and S is a statement. Then*

$$[\mathbf{var}\ v; S] = (\bigwedge_{d} \langle d/v \rangle); S; (\bigwedge_{d} \langle d/v \rangle)$$

Proof. For an arbitrary Q we have

$$((\bigwedge_{d} \langle d/v \rangle); S; \bigwedge_{d} \langle d/v \rangle)(Q)$$

$=$ [definitions of constructors]

$$(\bigwedge_{d} \langle d/v \rangle)(S(\bigwedge_{d} \langle d/v \rangle (Q)))$$

$=$ [Lemma 5(b)]

$$(\bigwedge_{d} \langle d/v \rangle)(S(\forall v \cdot Q))$$

$$= [\text{Lemma 5(b)}]$$

$$\forall v \cdot S(\forall v \cdot Q). \quad \square$$

Sequential and conditional composition. By definition, sequential composition in the specification language is the same as sequential composition in $\mathscr{C}^{\perp}$.

In order to construct conditional composition we first construct the assert statement $\{P\}$.

Lemma 16. *Assume that P is a predicate formula and that $v = Var(P)$. Then*

$$\{P\} = \bigvee_{d:P} \langle v = d \rangle.$$

Proof. Immediate from Lemma 4(a). $\square$

Conditional composition is now constructed in the following way (a similar constructions is used in [15]).

Theorem 10. *Assume that b_1 and b_2 are predicate formulas and S_1 and S_2 statements. Then*

$$\mathbf{if}\, b_1 \rightarrow S_1 \,\square\, b_2 \rightarrow S_2 \,\mathbf{fi} = \{b_1 \vee b_2\}; ((\{\neg b_1\}; \mathbf{serve} \vee S_1) \wedge (\{\neg b_2\}; \mathbf{serve} \vee S_2)).$$

Proof. If $Q = false$ then the equality follows from the strictness of all the statements involved. Now assume that $Q \neq false$. Then

$$(\{b_1 \vee b_2\}; ((\{\neg b_1\}; \mathbf{serve} \vee S_1) \wedge (\{\neg b_2\}; \mathbf{serve} \vee S_2)))(Q)$$

$$= [\text{definitions of constructors}]$$

$$\{b_1 \vee b_2\}((\{\neg b_1\}(\mathbf{serve}(Q)) \vee S_1(Q)) \wedge (\{\neg b_2\}(\mathbf{serve}(Q)) \vee S_2(Q)))$$

$$= [\text{definition of } \mathbf{serve}]$$

$$\{b_1 \vee b_2\}((\{\neg b_1\}(true) \vee S_1(Q)) \wedge (\{\neg b_2\}(true) \vee S_2(Q)))$$

$$= [\text{definition of assert statement; } true \text{ is top element of } Pred]$$

$$(b_1 \vee b_2) \wedge (\neg b_1 \vee S_1(Q)) \wedge (\neg b_2 \vee S_2(Q))$$

$$= [\text{predicate calculus}]$$

$$(b_1 \vee b_2) \wedge (b_1 \Rightarrow S_1(Q)) \wedge (b_2 \Rightarrow S_2(Q)). \quad \square$$

Recursive composition. Assume that $T(X)$ is an expression built from the symbol X, the primitive statements, sequential composition, meet and join. Then if X is replaced by any statement S of $Stat^{\perp}$, the result $T(S)$ will be another statement of $Stat^{\perp}$ and it can be constructed as a command in $\mathscr{C}^{\perp}$. Since the command constructors of $\mathscr{C}^{\perp}$ are monotonic in their arguments, $T(X)$ as a function of X is a monotonic function from $\mathscr{C}^{\perp}$ to $\mathscr{C}^{\perp}$. Since $\mathscr{C}^{\perp}$ is a complete lattice, T has a least fixpoint in $\mathscr{C}^{\perp}$. This least fixpoint equals $\mu X \cdot T(X)$.

This construction is generalized to the case when the expression $T(X)$ itself contains recursive compositions, by an inductive argument. Note that there is no need to introduce an explicit recursion constructor into the base language, as $\mu X \cdot T(X)$ is just an abbreviation for a command that already has a representation in terms of the primitive commands and the three constructors sequential composition, meet and join.

Derived statements. We already showed how the assert statement was constructed. Now we consider the multiple assignment statement $v:=e$.

Lemma 17. *Assume v is a list of variables and e is an expression. Then*

$$v:=e = \bigvee_d (\langle v=d\rangle; v\cdot(v=e[d/v])).$$

Proof. Follows directly from Theorem 7, using the definition of the multiple assignment statement $v:=e$ as $v:=v'\cdot(v'=e)$. $\square$

As an example, let us look at the simple assignment statement $u:=u+1$. Using Lemma 17, Theorem 7 and Lemma 12 we get (after simplifications)

$$u:=u+1 = \bigvee_c (\langle u=c\rangle; \langle c+1/u\rangle).$$

We can consider the substitution command $\langle c+1/u\rangle$ as the assignment while the test command $\langle u=c\rangle$ in combination with the join finds the correct value to be incremented. Gardiner and Morgan [11] use their conjunction operation (corresponding to our join) in this way to avoid having to mention the initial value of a variable in the postcondition.

9. Derivation of a Language with Miracles

In this section we introduce a specification language, denoted *Stat*, which extends the language of Sect. 8 by permitting miraculous statements. This language is syntactic and finitary (although permitting unbounded nondeterminism), in the same way as the language *Stat*$^{\perp}$. In addition, it is self-dual, in the sense that the dual of every statement in *Stat* is also in *Stat*.

9.1. The Specification Language

We shall refer to the language *Stat* as *the extended specification language.* The language *Stat* has two primitive statements, the *demonic, miraculous assignment statement* and the *angelic, strict assignment statement.* As statement constructors we have sequential composition, meet and join and two fixpoint compositions. The statements of *Stat* are defined by

$S::= v:\approx v'\cdot P$	*(demonic, miraculous assignment statement)*
$\mid\ \overline{v:=v'\cdot P}$	*(angelic, strict assignment statement)*
$\mid\ X$	*(statement variable)*
$\mid\ S_1; S_2$	*(sequential composition)*
$\mid\ S_1 \wedge S_2$	*(meet, demonic composition)*
$\mid\ S_1 \vee S_2$	*(join, angelic composition)*
$\mid\ \mu X\cdot T$	*(least fixpoint composition)*
$\mid\ \nu X\cdot T$	*(greatest fixpoint composition).*

Here v is a list of variables, b is a predicate and S, S_1 and S_2 are statements. In the assignment statements, v' is a list of auxiliary variables, not belonging to *Var*, and P is a predicate formula.

Note that we do not define the block or the conditional composition. We shall see later how these can be defined using the other constructs.

Meaning of the statements. We now give the statements meanings as predicate transformers. The angelic strict assignment statement (see (22)) and sequential composition have the same meanings as in $Stat^{\perp}$, while meet and join have the same meaning as in the base language $\mathscr{C}^{\perp}$. The meaning of the demonic, miraculous assignment statement differs from the demonic assignment statement of $Stat^{\perp}$ in the following respect: $v{:}{\approx}v'\cdot P$ succeeds miraculously if it is not possible to assign values v' to v so that the condition P becomes established (in this situation, $v{:=}v'\cdot P$ aborts).

Thus the meanings are as follows:

$$(v{:}{\approx}v'\cdot P)(Q)\overset{\text{def}}{=}\forall v'\cdot(P\Rightarrow Q[v'/v])$$

$$(\overline{v{:=}v'\cdot P})(Q)\overset{\text{def}}{=}\exists v'\cdot(P\wedge Q[v'/v])$$

$$(S_1;S_2)(Q)\overset{\text{def}}{=}S_1(S_2(Q))$$

$$(S_1\wedge S_2)(Q)\overset{\text{def}}{=}S_1(Q)\wedge S_2(Q))$$

$$(S_1\vee S_2)(Q)\overset{\text{def}}{=}S_1(Q)\vee S_2(Q)).$$

The fixpoint compositions are given the following meanings: $\mu X\cdot T(X)$ is the least and $\nu X\cdot T(X)$ the greatest fixpoint of the monotonic function $\lambda X\cdot T(X)$ on the complete lattice of predicate transformers. The existence of a least as well as a greatest fixpoints is guaranteed by the Knaster-Tarski fixpoint theorem. Note that the least fixpoint composition in *Stat* corresponds to recursive composition in $Stat^{\perp}$.

Monotonicity and substatement monotonicity. All statements of $Stat^{\perp}$ were monotonic (considered as predicate transformers) and all constructors of $Stat^{\perp}$ were shown to be monotonic with respect to substatement replacement. The following lemma shows that the same is true for *Stat*.

Lemma 18. *All statements of the extended specification language are monotonic and all statement constructors are monotonic with respect to substatement replacement.*

Duals in the extended specification language. Let the duality operator $(.)^0$ be defined by (14), i.e. $S^0(Q)=\neg S(\neg Q)$ for all Q. Then the dual of every statement in *Stat* can be constructed using the following lemma.

Lemma 19. *The following dualities hold in Stat*:
(a) $(v{:}{\approx}v'\cdot P)^0=\overline{v{:=}v'\cdot P}$
(b) $(S_1;S_2)^0=S_1^0;S_2^0$
(c) $(S_1\wedge S_2)^0=S_1^0\vee S_2^0$
(d) $(\mu X\cdot T(X))^0=\nu X\cdot T^0(X)$

where the expression $T^0(X)$ is defined as

$$T^0(S) \overset{\text{def}}{=} (T(S^0))^0$$

for all statements S in Stat.

Proof. We prove (a) here. Parts (b) and (c) follow from Lemma 6 and part (d) is proved in the Appendix.

$(v:\approx v' \cdot P)^0 (Q)$

$=$ [definition of assignments, duals]

$\neg\forall v' \cdot (P \Rightarrow \neg Q[v'/v])$

$=$ [calculus]

$\exists v' \cdot \neg(\neg P \vee \neg Q[v'/v])$

$=$ [calculus]

$\exists v' \cdot (P \wedge Q[v'/v])$

$=$ [definition of assignments]

$\overline{v:=v' \cdot P}$. $\square$

Thus the language has a strong sense of duality. Note that we do not include the duality operator as a real statement constructor in *Stat*. It has the disadvantage of not being monotonic with respect to substatement replacement. Furthermore, the above result shows that it is not needed.

Derived statements. As in *Stat*$^\perp$, we define derived statements in *Stat*. Let ε be the empty list and let P be an arbitrary predicate formula. Then we define the *assert statement* $\{P\}$ and its dual, the *guard statement* $P \rightarrow$ by

$$\{P\} \overset{\text{def}}{=} \overline{\varepsilon:=\varepsilon \cdot P},$$

$$P \rightarrow \overset{\text{def}}{=} \varepsilon :\approx \varepsilon \cdot P$$

(the guard statement has been introduced in Hesselink [15]). This gives the following meanings:

$$\{P\}(Q) = P \wedge Q,$$
$$(P \rightarrow)(Q) = P \Rightarrow Q.$$

As special cases, we have $\{\mathit{false}\} =$ **abort** and $(\mathit{true} \rightarrow) =$ **magic** (note that *Stat* is a lattice with a top and a bottom element, though it is not complete).

We can now define the *guarded statement* $P \rightarrow S$ to be an abbreviation for the composition $(P \rightarrow); S$. Thus,

$$(P \rightarrow S)(Q) = P \Rightarrow S(Q)$$

which corresponds to the definitions given in [4, 18, 21]. Note that the dual of $P \rightarrow$ is $\{P\}$, and that we also may use the abbreviation $\{P\}\,S$ for $\{P\}; S$.

Now let v be a list of program variables, v' a list of auxiliary variables and P a predicate formula. Then

$(\overline{v:=v'\cdot P}; \mathbf{magic} \wedge v:\approx v'\cdot P)(Q)$

$=$ [definition of constructors]

$(\overline{v:=v'\cdot P}(\mathbf{magic}(Q)) \wedge (v:\approx v'\cdot P)(Q)$

$=$ [definition of **magic**]

$(\overline{v:=v'\cdot P}(\mathit{true}) \wedge (v:\approx v'\cdot P)(Q)$

$=$ [definition of assignment statements]

$\exists v'\cdot P \wedge \forall v'\cdot(P \Rightarrow Q[v'/v])$.

Thus the *demonic strict assignment statement* (i.e. the demonic assignment statement (21) of the original specification language $\mathit{Stat}^{\perp}$ can be defined as

$$v:=v'\cdot P \overset{\mathrm{def}}{=} \overline{v:=v'\cdot P}; \mathbf{magic} \wedge v:\approx v'\cdot P.$$

Dually, we can define the *angelic miraculous assignment statement* as

$$\overline{v:\approx v'\cdot P} \overset{\mathrm{def}}{=} v:\approx v'\cdot P; \mathbf{abort} \vee \overline{v:=v'\cdot P}$$

Using the demonic strict assignment statement we can define the ordinary multiple assignment statement the same way as in $\mathit{Stat}^{\perp}$.

We now show that the block statement and the conditional composition, as defined in $\mathit{Stat}^{\perp}$, can be constructed in *Stat*.

Let v be any list of variables and S any statement in the extended specification language. Then

$((v:=v\cdot \mathit{true}); S; (v:=v\cdot \mathit{true}))(Q)$

$=$ [definition of sequential composition]

$(v:=v\cdot \mathit{true})(S((v:=v\cdot \mathit{true})(Q)))$

$=$ [definition of demonic assignment statement]

$(v:=v\cdot \mathit{true})(S(\forall v\cdot Q))$

$=$ [definition of demonic assignment statement]

$\forall v\cdot S(\forall v\cdot Q)$

for all predicates Q. Thus we can define the block statement as

$$[\mathbf{var}\ v; S] \overset{\mathrm{def}}{=} v:=v\cdot \mathit{true}; S; v:=v\cdot \mathit{true}.$$

This definition clearly shows the intuitive meaning of our block construct; the value of v is undefined when entering the block as well as when leaving it.

Now let b_1 and b_2 be any predicates and S_1 and S_2 any statements in the extended specification language. Then

$(\{b_1 \vee b_2\}; (b_1 \rightarrow S_1 \wedge b_2 \rightarrow S_2))(Q)$

= [definition of constructors]

$\{b_1 \vee b_2\}\ ((b_1 \to S_1)(Q) \wedge (b_2 \to S_2)(Q))$

= [definition of assert statement and guarded statement]

$(b_1 \vee b_2) \wedge (b_1 \Rightarrow S_1(Q)) \wedge (b_2 \Rightarrow S_2(Q))$

for all predicates Q.

Thus we can define conditional composition, in *Stat* called *demonic conditional composition* as

$$\textbf{if}\, b_1 \to S_1 \,\square\, b_2 \to S_2 \,\textbf{fi} \overset{\text{def}}{=} \{b_1 \vee b_2\}; (b_1 \to S_1 \wedge b_2 \to S_2).$$

Identifying the symbols $\square$ and $\wedge$ (both standing for a demonic choice) we could instead have defined **if**...**fi** as a constructor, using the definition of [19, 21],

$$(\textbf{if}\, S\, \textbf{fi})(Q) = \neg S(\mathit{false}) \wedge S(Q). \tag{25}$$

We have two good reasons for *not* doing this. The first reason is that this definition gives the traditional meaning to the conditional composition **if** $b_1 \to S_1 \square b_2 \to S_2$ **fi** only if S_1 and S_2 are strict (non-miraculous). However, this problem can be overcome. As noted in [19], any miraculous statement S can be written as $b' \to S'$ where S' is strict. Since $b \to (b' \to S') = (b \wedge b') \to S'$, we can always rewrite a conditional composition into a form **if** $b'_1 \to S'_1 \square b'_2 \to S'_2$ **fi** where S'_1 and S'_2 are strict.

Our second and stronger objection to the constructor **if**...**fi** is that it is not monotonic with respect to substatement replacement. This limits its usefulness in program development, as pointed out by Morgan [19]. As we shall see, this also means that it cannot be defined using the constructors of our extended base language. Hence, we will not accept (25) as a definition.

We define a second conditional composition;

$$\textbf{if}\, b_1 \to S_1 \diamond b_2 \to S_2 \,\textbf{fi} \overset{\text{def}}{=} (\{b_1\}; S_1) \vee (\{b_2\}; S_2).$$

This is the *angelic conditional composition.* When executed in a state where both b_1 and b_2 holds, it makes an angelic choice between S_1 and S_2. In all other states it acts like the ordinary conditional composition.

When there is only one alternative, the two conditional compositions coincide; we define

$$\textbf{if}\, b \to S \,\textbf{fi} \overset{\text{def}}{=} \{b\}; S.$$

9.2. *Construction of the Specification Language Statements in $\mathscr{C}$*

We now show how all the statements in the extended specification language can be constructed within the extended base language $\mathscr{C}$. The angelic strict assignment statement is constructed as in Sect. 8. Sequential composition, meet and join are the same as the corresponding constructors in $\mathscr{C}$.

From the duality shown above it follows that the construction of the demonic miraculous assignment statement is the dual of the construction of the angelic assignment statement. Thus

$$v :\approx v' \cdot P = \bigwedge_{d} (\langle v = d \rangle^{0} ; (v \sim P[d, v/v, v']))$$

using the *demonic miraculous update statement* $v \sim P$, constructed as

$$v \sim P = \bigwedge_{d:P} \langle d/v \rangle$$

when $Var(P) \subseteq v$, and in the general case as

$$v \sim P = \bigwedge_{d} (\langle w = d \rangle^{0} ; v \sim P[d/w])$$

where the list w consists of all the variables that P refers to and which are not in v.

Since every monotonic predicate transformer can be constructed in $\mathscr{C}$, all fixpoint statements have a construction in $\mathscr{C}$.

Derived statements. The derived statements were defined in terms of primitive statements and the constructors ";", " $\wedge$ " and " $\vee$ ". Since the primitive statements have been constructed in $\mathscr{C}$, we can also construct the derived statements. In particular, we have the two dual constructions

$$\{P\} = \bigvee_{d:P} \langle v = d \rangle,$$

$$P \rightarrow = \bigwedge_{d:P} \langle v = d \rangle^{0}.$$

The suggested constructor **if**...**fi** cannot be constructed within $\mathscr{C}$. This is because all the constructors of $\mathscr{C}$ are monotonic with respect to substatement replacement but **if**...**fi** is not. This does not mean that we cannot construct every separate statement **if** S **fi**. Rather, it means that there is no expression $T(X)$ built up of the primitive statements and constructors of $\mathscr{C}$ in addition to the statement variable X, such that $T(S) = \textbf{if}\, S\, \textbf{fi}$ for all statements S. By the same argument, we can show that the duality operator $(\ )^{0}$ cannot be constructed within $\mathscr{C}$.

10. Conclusion

We have shown how every strict monotonic predicate transformer can be constructed within a lattice-based command language $\mathscr{C}^{\perp}$ with very simple primitive commands. We extended the base language $\mathscr{C}^{\perp}$ to cover all monotonic predicate transformers by combining it with a dual language $\mathscr{C}^{\top}$. This dual language was constructed by changing one of the primitive statements of $\mathscr{C}^{\perp}$ from strict to miraculous. This strict base language $\mathscr{C}^{\perp}$ was shown to be complete, in the sense that all strict monotonic predicate transformers can be constructed within it. Similarly, the extended base language was shown to be complete, in the sense that it contains all monotonic predicate transformers.

The weakest precondition predicate transformers for the statements of a quite general specification language, permitting both demonic and angelic nondeterminism, were then constructed within $\mathscr{C}^{\perp}$.

Finally we defined a general specification language with a strong sense of duality, permitting both demonic and angelic nondeterminism as well as strict and miraculous statements, and showed how the statements of this language can be constructed using the extended base language $\mathscr{C}$.

Acknowledgements. We have benefitted greatly from discussions with Carroll Morgan, John Tucker and Viking Högnäs. This research was funded by the Academy of Finland and the Finsoft III research project by Tekes. We also wish to thank the referees for their valuable comments and suggestions, which lead to a number of improvements.

Appendix: Proofs of Lemmas

Proof of Lemma 2. Let σ be an arbitrary state. Then

$$(\bigvee_{d}((v=d)\wedge P[d/v]))(\sigma)=tt$$

$\Leftrightarrow$ [$\vee$, $\wedge$ as pointwise extension]

$$\bigvee_{d}((v=d)(\sigma)\wedge P[d/v](\sigma))=tt$$

$\Leftrightarrow$ [join in *Bool* is logical disjunction]

$((v=d)(\sigma)=tt)\wedge(P[d/v](\sigma)=tt)$, for some d

$\Leftrightarrow$ [definitions of $(v=d)$ and $P[d/v]$]

$(\sigma(v)=d)\wedge(P(\sigma[d/v])=tt)$, for some d

$\Leftrightarrow$ [one-point rule]

$P(\sigma[\sigma(v)/v])=tt)$

$\Leftrightarrow$

$P(\sigma)=tt.$

Proof of Lemma 4. First, we prove (a).

$$(\bigvee_{d:P}\langle v=d\rangle)(Q)$$

$=$ [definition of qualified join]

$(\bigvee(d: P[d/v]=true: \langle v=d\rangle))(Q)$

$=$ [$\vee$ as pointwise extension]

$\bigvee(d: P[d/v]=true: \langle v=d\rangle(Q))$

$=$ [definition of $\langle v=d\rangle$]

$\bigvee(d: P[d/v] = true: (v = d) \wedge Q)$

$=$ [Join Infinite Distributivity, Lemma 3]

$\bigvee(d: P[d/v] = true: (v = d)) \wedge Q$

$=$ [definition of qualified join]

$(\bigvee_{d:P} (v = d)) \wedge Q$

$=$ [Lemma 1]

$P \wedge Q$.

Now we move to (b).

$(\bigwedge_{d:P} \langle d/v \rangle)(Q)$

$=$ [definition of qualified join, pointwise extension]

$\bigwedge(d: P[d/v] = true: \langle d/v \rangle(Q))$

$=$ [definition of $\langle d/v \rangle$]

$\bigwedge(d: P[d/v] = true: Q[d/v])$.

Now let σ be an arbitrary state. Then

$(\bigwedge(d: P[d/v] = true: Q[d/v]))(\sigma) = tt$

$\Leftrightarrow$[$\wedge$ is defined as pointwise extension]

$\bigwedge(d: P[d/v] = true: Q[d/v](\sigma)) = tt$

$\Leftrightarrow$[$\wedge$ in *Bool* is logical conjunction]

$(P[d/v] = true) \Rightarrow (Q[d/v](\sigma) = tt)$, for all lists d

$\Leftrightarrow$[$Var(P) \subseteq v$ implies that $P[d/v]$ is *true* or *false*]

$(P[d/v](\sigma) = tt) \Rightarrow (Q[d/v](\sigma) = tt)$, for all lists d

$\Leftrightarrow$[definition of $\Rightarrow$ in *Bool*]

$(P[d/v](\sigma) \Rightarrow Q[d/v](\sigma)) = tt$, for all lists d

$\Leftrightarrow$[substitution distributes over connectives]

$(P \Rightarrow Q)[d/v](\sigma) = tt$, for all lists d

$\Leftrightarrow$[definition of quantified predicate]

$(\forall v \cdot (P \Rightarrow Q))(\sigma) = tt$.

Thus, (b) is proved. The proof of (c) proceeds exactly as the proof of (b).

Proof of Lemma 6. Let Q be an arbitrary predicate. First, we prove (a).

$(S^0)^0(Q)$

$=$ [definition of dual]

$\neg S^0(\neg Q)$

$=$ [definition of dual]

$\neg(\neg S(\neg\neg Q))$

$=$ [property of inverse in the boolean lattice *Pred* (double negation)]

$S(Q)$

Next, we prove (b).

S^0 monotonic

$\Leftrightarrow$ [definition of monotonicity]

$\forall Q, Q' \cdot (Q \leqq Q' \Rightarrow S^0(Q) \leqq S^0(Q'))$

$\Leftrightarrow$ [definition of dual]

$\forall Q, Q' \cdot (Q \leqq Q' \Rightarrow \neg S(\neg Q) \leqq \neg S(\neg Q'))$

$\Leftrightarrow$ [property of inverse in the boolean lattice *Ptran*]

$\forall Q, Q' \cdot (Q \leqq Q' \Rightarrow S(\neg Q') \leqq S(\neg Q))$

$\Leftrightarrow$ [property of inverse in the boolean lattice *Pred*]

$\forall Q, Q' \cdot (\neg Q' \leqq \neg Q \Rightarrow S(\neg Q') \leqq S(\neg Q))$

$\Leftrightarrow$ [every element in a boolean lattice is the inverse of a unique element]

$\forall Q, Q' \cdot (Q' \leqq Q \Rightarrow S(Q') \leqq S(Q))$

$\Leftrightarrow$ [definition of monotonicity]

S monotonic.

Now, (c).

$(S_1^0; S_2^0)(Q)$

$=$ [definition of sequential composition]

$S_1^0(S_2^0(Q))$

$=$ [definition of dual]

$S_1^0(\neg S_2(\neg Q))$

$=$ [definition of dual]

$\neg S_1(\neg \neg S_2(\neg Q))$

$=$ [property of inverse in the boolean lattice *Pred* (double negation)]

$\neg S_1(S_2(\neg Q)$

$=$ [definition of sequential composition]

$\neg (S_1; S_2)(\neg Q)$

$=$ [definition of dual]

$(S_1; S_2)^0(Q)$.

Next, (d).

$S_2^0 \leqq S_1^0$

$\Leftrightarrow$ [definition of partial order in *Ptran*]

$\forall Q \cdot S_2^0(Q) \leqq S_1^0(Q)$

$\Leftrightarrow$ [definition of dual]

$\forall Q \cdot \neg S_2(\neg Q) \leqq \neg S_1(\neg Q)$

$\Leftrightarrow$[property of inverse in the boolean lattice *Pred*]

$\forall Q \cdot S_1(\neg Q) \leqq S_2(\neg Q)$

$\Leftrightarrow$[every element of a boolean lattice is the inverse of a unique element]

$\forall Q \cdot S_1(Q) \leqq S_2(Q)$

$\Leftrightarrow$[definition of the partial order in *Ptran*]

$S_1 \leqq S_2$.

Then, (e).

$(\bigwedge S_i)^0(Q)$

$=$[definition of dual]

$\neg(\bigwedge S_i)(\neg Q)$

$=$[DeMorgan rule (1) for complete boolean lattices]

$\bigvee(\neg S_i(\neg Q))$

$=$[definition of dual]

$\bigvee S_i^0(Q)$

Now, (f) is proved using (a) and (e). To prove (g), assume that S is conjunctive and let P and Q be arbitrary predicates. Then

$S^0(P \vee Q)$

$=$[definition of dual]

$\neg S(\neg(P \vee Q))$

$=$[DeMorgan rule (1) for predicates]

$\neg S(\neg Q \wedge \neg P)$

$=$[S is conjunctive]

$\neg(S(\neg Q) \wedge S(\neg P))$

$=$[DeMorgan rule (2) for predicates]

$\neg S(\neg Q) \vee \neg S(\neg P)$

$=$[definition of dual]

$S^0(P) \vee S^0(Q)$.

Finally, (h) is proved using (a) and (g).

Proof of Lemma 7. Let Q be an arbitrary predicate. Then

$(\bigvee_{c' \neq c}((u=c'); \textbf{magic}) \vee \bigvee_{c'}(u=c'))(Q)$

$=$[definitions of constructors]

$\bigvee_{c' \neq c}((u=c')(\textbf{magic}(Q))) \vee \bigvee_{c'}((u=c')(Q))$

$=$[definition of **magic**, Lemma 5(a)]

$\bigvee_{c' \neq c} (\langle u=c' \rangle (true)) \vee Q$

$=$ [definition of test statement]

$\bigvee_{c' \neq c} (u=c') \vee Q$

$=$ [predicate calculus]

$\neg (u=c) \vee Q$

$=$ [predicate calculus]

$(u=c) \Rightarrow Q$.

The second part is proved in the same way.

Proof of Lemma 18. The monotonicity of the two primitive statements is straightforward to prove. It is equally straightforward to prove that the constructors preserve monotonicity.

Substatement monotonicity of sequential composition, meet and join follow from the corresponding results of earlier sections.

Substatement monotonicity for least and greatest fixpoint composition is proved as for recursive composition. First assume that $T(X)$ is an expression built up of primitive statements, sequential composition, meet and join. Further assume that replacing some substatement of $T(X)$ (except X) with a greater statement gives the new expression $T'(X)$. By the substatement monotonicity of sequential composition, meet and join, we have

$$T(S) \leqq T'(S) \qquad \text{for all statements } S. \tag{26}$$

Let μ and μ' stand for the least fixpoints of $\lambda X \cdot T(X)$ and $\lambda X \cdot T(X)$, respectively. By the definition of least fixpoints we know that

$$T(\mu) = \mu \tag{27}$$

$$T(S) \leqq S \Rightarrow \mu \leqq S \qquad \text{for all } S \tag{28}$$

$$T'(\mu') = \mu' \tag{29}$$

$$T'(S) \leqq S \Rightarrow \mu' \leqq S \qquad \text{for all } S \tag{30}$$

Thus, by (26) and (29),

$$T(\mu') \leqq T'(\mu') = \mu'$$

and by (28),

$$\mu \leqq \mu'$$

Dually, by the definition of greatest fixpoints we know that

$$T(\nu) = \nu \tag{31}$$

$$S \leqq T(S) \Rightarrow S \leqq \nu \qquad \text{for all } S \tag{32}$$

$$T'(\nu') = \nu' \tag{33}$$

$$S \leqq T'(S) \Rightarrow S \leqq \nu' \qquad \text{for all } S \tag{34}$$

where v and v' stand for the greatest fixpoints of $\lambda X \cdot T(X)$ and $\lambda X \cdot T(X)$, respectively. Thus, by (31) and (26),

$$v = T(v) \leqq T'(v)$$

and by (34),

$$v \leqq v'.$$

We have thus shown substatement monotonicity for least and greatest fixpoint composition, in the case when T contains no fixpoint composition. The case when T contains one or more fixpoint compositions is now proved by an inductive argument.

Proof of Lemma 19(d). Let μ be $\mu X \cdot T(X)$. Then we have to show that μ^0 is the greatest fixpoint of T^0, i.e. that the following two conditions hold:

$$T^0(\mu^0) = \mu^0$$
$$S \leqq T^0(S) \Rightarrow S \leqq \mu^0 \qquad \text{for all } S.$$

We have

$T^0(\mu^0)$
$=$ [definition of T^0]
$(T((\mu^0)^0))^0$
$=$ [Lemma 6(a)]
$(T(\mu))^0$
$=$ [μ is fixpoint of T]
μ^0

and for arbitrary S,

$S \leqq T^0(S)$
$\Rightarrow$ [definition of T^0]
$S \leqq (T(S^0))^0$
$\Rightarrow$ [Lemma 6]
$T(S^0) \leqq S^0$
$=$ [μ is least fixpoint of T]
$\mu \leqq S^0$
$\Rightarrow$ [Lemma 6]
$S \leqq \mu^0$.

References

1. 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

2. Back, R.J.R.: Correctness preserving program refinements: Proof theory and applications. Mathematical Centre Tracts 131, Mathematical Centre, Amsterdam 1980
3. Back, R.J.R.: A calculus of refinements for program derivations. Acta Inf. **25**, 593–624 (1988)
4. Back, R.J.R.: Refining atomicity in parallel algorithms. Reports on Computer Science and Mathematics 57, Åbo Akademi 1988 (To appear in Conference on Parallel Architectures and Languages Europe 1989)
5. Back, R.J.R.: Changing data representation in the refinement calculus. Hawaii International Conference on System Sciences 1989 (also available as Data Refinement in the Refinement Calculus, Reports on Computer Science and Mathematics 68, Åbo Akademi 1988)
6. de Bakker, J.: Mathematical theory of program correctness. Englewood Cliffs, NJ: Prentice-Hall 1980
7. Birkhoff, G.: Lattice theory. Providence, RI: American Mathematical Society 1961
8. Broy, M.: A theory for nondeterminism, parallellism, communications and concurrency. Theor. Comput. Sci. **45**, 1–61 (1986)
9. Dijkstra, E.W.: A discipline of programming. Englewood Cliffs, NJ: Prentice Hall 1976
10. Dijkstra, E.W., van Gasteren, A.J.M.: A simple fixpoint argument without the restriction to continuity. Acta Inf. **23**, 1–7 (1986)
11. Gardiner, P., Morgan, C.: Data refinement of predicate transformers. Theor. Comput. Sci. (To appear)
12. Grätzer, G.: General lattice theory. Basel: Birkhäuser 1978
13. Hehner, E.: **do** considered **od**: A contribution to the programming calculus. Acta Inf. **11**, 287–304 (1979)
14. Hehner, E.: The logic of programming. Englewood Cliffs, NJ: Prentice-Hall 1984
15. Hesselink, W.H.: An algebraic calculus of commands. CS 8808, Department of Mathematics and Computer Science, University of Groningen 1988
16. Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, A.: Laws of programming. Commun. ACM **30**, 672–686 (1987)
17. Jacobs, D., Gries, D.: General correctness: A unification of partial and total correctness. Acta Inf. **22**, 67–83 (1985)
18. Morgan, C.: Data refinement by miracles. Inf. Process. Lett. **26**, 243–246 (1988)
19. Morgan, C.: The specification statement. ACM TOPLAS **10**, 403–419 (1988)
20. Morris, J.: A theoretical basis for stepwise refinement and the programming calculus. Sci. Progr. **9**, 287–306 (1987)
21. Nelson, G.: A generalization of Dijkstra's calculus. Tech. Rep. 16, Digital Systems Research Center, Palo Alto, Calif. 1987
22. Park, D.: On the semantics of fair parallelism. Lect. Notes Comput. Sci. **86**, 504–526 (1980)
23. Smyth, M.B.: Powerdomains. J. Comput. Syst. Sci. **16**, 23–36 (1978)
24. Tarski, A.: A lattice theoretical fixed point theorem and its applications. Pac. J. Math. **5**, 285–309 (1955)