An Operational Semantics for the Guarded Command Language *

Johan J. Lukkien

Eindhoven University of Technology
Computer Science
P.O. Box 513
5600 MB Eindhoven, The Netherlands

Abstract. In [6], Dijkstra and Scholten present an axiomatic semantics for Dijkstra's guarded command language through the notions of weakest precondition and weakest liberal precondition. The informal notion of a computation is used as a justification for the various definitions. In this paper we present an operational semantics in which the notion of a computation is made explicit. The novel contribution is a generalization of the notion of weakest precondition. This generalization supports reasoning about general properties of programs (i.e, not just termination in a certain state).

1 Introduction

In [6], Dijkstra and Scholten present an axiomatic semantics for Dijkstra's guarded command language through the notions of weakest precondition and weakest liberal precondition. The informal notion of a computation is used as a justification for the various definitions. In this paper we present an operational semantics in which a computation is defined explicitly as a sequence of states through which execution of the program may evolve, starting from a certain initial state. This way of characterizing program executions is fairly conventional. Earlier work on the topic may be found in [8]; in [1] such a characterization is used to define liveness and safety formally. These state sequences play an important rôle in temporal logic (e.g., [10]).

The contributions of this paper are twofold. The way repetition is dealt with differs a little from the usual approach. Repetition has been defined both as the solution of a recursive equation and as the limit of its unfoldings. In this paper we also use a definition in terms of the limit of unfoldings but we explicitly locate limit series in these unfoldings such as to include unbounded choice in an easy way. Then we show that the repetition is also a solution of a recursive equation. This equation is used to prove that the weakest precondition of the repetition is the strongest solution of a recursive equation and that the weakest liberal precondition is the weakest solution of a recursive equation. The main contribution of this paper is a generalization of the notion of weakest precondition. This generalization supports reasoning about general properties of programs (i.e, not just termination in a certain state), including temporal properties. The alternative definition of the repetition is

* The research described in this paper was conducted while the author was on leave from Groningen University at the California Institute of Technology.

especially helpful in determining which extreme solution of a recursive equation to choose as the weakest precondition of the repetition, for such a property.

We use the term *operational* for this semantics because it determines the states that some abstract mechanism executing the program goes through. Sometimes this term is reserved for a labelled transition system in which the atomic actions of the program are the labels for transitions between states. A semantics like ours is then usually called *denotational*.

This paper proceeds as follows. In the next section we introduce the language and its operational semantics. In the third section we introduce properties of programs and the generalized notion of weakest precondition; we discuss Dijkstra's *wp* and *wlp* in detail. We also generalize the notion of (non)determinate programs. We show how to introduce the temporal property *ever* (or *eventually*, the operator "$\diamond$" from temporal logic). We end with some conclusions.

2 Operational Semantics

In defining the operational semantics of our language we define what happens if a program written in that language is executed by a machine. The machine has an internal state that is changed according to the program's instructions. This state consists of the values of the variables declared in the program. The semantics of a program is the set of all state sequences (finite or infinite) that may result from executing the program. We call these sequences *computations*. For our purposes it is sufficient to restrict our attention to this state, the space in which it assumes its values and the sequences over this space.

We first introduce some notation.

V = The list of all program variables.
X = A cartesian product of domains, one for each program variable, in the order in which they occur in V. X is called the state space and its elements are called states. X is nonempty.
$Bool$ = $\{true, false\}$.
P = The functions from X to $Bool$. Elements of P are called predicates.
T = The set of all nonempty finite or infinite sequences of states.

For $t \in T$, we use t^∞ to denote an infinite sequence of t's. For strings, we use juxtaposition to denote catenation. This is extended to sets in the obvious way. The catenation of a sequence to the tail of an infinite sequence is that infinite sequence. For a string s, $|s|$ denotes its length, $s.i$ element i if $0 \leq i < |s|$ and $last.s$ its last element if $|s| < \infty$. For strings s and t, $s \sqsubseteq t$ denotes that s is a prefix of t. $\sqsubseteq$ is a partial order on strings. Sometimes it is convenient to distinguish between finite and infinite strings. For A a set of strings, $fin.A$ is the subset of A consisting of the finite strings in A and $inf.A$ the remaining part of A.

P, equipped with the pointwise implication order is a boolean lattice (see, for instance, [5]). We use $[p]$ as a shorthand for $\forall(x : x \in X : p.x)$. Hence, the fact that p precedes q in the order is denoted by $[p \Rightarrow q]$. We use the name *true* for the function that yields *true* everywhere and *false* for the function that yields *false* everywhere.

With these preliminaries, we define programs. A program describes which sequences are possible computations, hence a program may be viewed as a subset of T. However, not all these subsets can be programs. If a subset of T is to be viewed as a program there must be, for every $x \in X$, a sequence in that subset that starts with x, since execution of a program can commence in an arbitrary initial state.

$$Prog = \{S : S \subseteq T \wedge \forall(x : x \in X : \exists(s : s \in S : s.0 = x)) : S\},$$
the set of programs.

So, we do not bother whether a program is a computable set and indeed, many elements of *Prog* are not computable. We only have the restriction with respect to initial states. Notice that since X is nonempty, a program can never be the empty set.

We now look more closely at our programming language. We list the constructs and their intended meaning.

abort	- loop forever
skip	- do nothing
$y := e$	- assign value e to program variable y
$S; U$	- sequential composition
if $[](i :: B_i \rightarrow S_i)$ **fi**	- execute an S_i for which B_i holds
do $B \rightarrow S$ **od**	- repeat S as long as B holds

All these constructs are defined as elements of *Prog*. For every definition we have to verify whether we defined a proper element of *Prog*, i.e., we have to verify the restriction in the definition of *Prog*.

We start with the definitions of the three basic constructs.

$$\begin{aligned} abort &= \{x : x \in X : x^{\infty}\} \\ skip &= X \end{aligned}$$

abort repeats the state in which it is started forever. *skip* does not do anything: initial state and final state coincide.

In order to define the assignment statement, we introduce substitution. Let d be a function from X to the component of the state space in which some program variable y assumes its values.

$$x[y/d.x].z = \begin{cases} x.z & \text{if } y \neq z \\ d.x & \text{if } y = z \end{cases}$$

Using this, the assignment statement is defined by

$$y := e = \{x : x \in X : x(x[y/e.x])\}.$$

The assignment changes the state in which it is started in that at position $y, e.x$ is stored. Clearly, all three constructs satisfy the constraint imposed in the definition of *Prog*.

We define sequential composition in a slightly more general context than only for members of *Prog*. We define it for general subsets of T. For $A, B \subseteq T$,

$$A; B = \{a, x, b : ax \in A \wedge xb \in B : axb\} \cup inf.A.$$

$A; B$ contains the infinite sequences in A and all sequences formed from a finite sequence $a \in A$ and a sequence $b \in B$ such that the last element of a is the first element of b, by catenating a without its last element, and b. In the case that B is nonempty, we can omit the term $inf.A$ because of our convention with respect to the catenation of infinite sequences.

Property 1. *Let* $A, B \subseteq T$.

$$(i)\ \forall(s : s \in A; B : \exists(a : a \in A : a \sqsubseteq s))$$
$$(ii)\ B \in Prog \Rightarrow \forall(a : a \in A : \exists(s : s \in A; B : a \sqsubseteq s))$$

Proof. (i) follows directly from the definition of ";". (ii) Choose $a \in A$. If $|a| = \infty$, $a \in A; B$. If $|a| < \infty$, there exists a $xb \in B, x \in X$ such that $a = cx$ since $B \in Prog$. From the definition of ";" it follows that $cxb \in A; B$. Obviously, $a \sqsubseteq cxb$.

From property 1, it follows that $A, B \in Prog$ implies $A; B \in Prog$.

In both the alternative construct and the repetition we have the notion of a guard. A guard is a predicate. Operationally, evaluation of a guard is the restriction to the states for which it yields *true*. We do want to record the evaluation of a guard such that an infinite number of evaluations yields an infinite sequence. So, for a predicate p we define

$$p^{guard} = \{x : x \in X \wedge p.x : xx\}.$$

Now we can define the alternative construct. We abbreviate **if** $[](i :: B_i \rightarrow S_i)$ **fi** by IF.

$$IF = \bigcup(i :: B_i^{guard}; S_i) \cup (\forall(i :: \neg B_i))^{guard}; abort$$

We have to show again that we defined a proper member of *Prog*. Notice that, for a predicate p, $p^{guard}; S$ is the restriction of S to those sequences that start with a state for which p holds, with the initial state repeated. Since for every $x \in X$ we have $\forall(i :: \neg B_i.x) \vee \exists(i :: B_i.x)$, the fact that IF is an element of *Prog* follows from the fact that S_i and *abort* are elements of *Prog*.

Finally, we define repetition. Executing the repetitive construct **do** $B \rightarrow S$ **od**, corresponds to the repeated, sequential execution of the body, S, possibly an infinite number of times. We therefore first study some properties of the sequential composition.

Property 2. *Let* $A, B \subseteq T$.

$$(i)\ inf.(A; B) = inf.A \cup (fin.A); inf.B$$
$$(ii)\ fin.(A; B) = (fin.A); fin.B$$

Proof. Immediate from the definition.

Property 3. *For $A, B, C \subseteq T$ such that A and B contain no infinite sequences,*

$$(A; B); C = A; (B; C).$$

Proof. Notice that, according to property $2(i)$, $inf.(A; B)$ is also empty. Choose $s \in (A; B); C$. By definition, s can be written as uxc such that $ux \in A; B$ and $xc \in C$. ux may be written in turn as $ayb, ay \in A, yb \in B$. If b is the empty string, we have $x = y$ and $x \in B$. Hence, $xc \in B; C$ and $s = axc \in A; (B; C)$. If b is not the empty string, we can write yb as $yb'x$ hence $yb'xc \in B; C$ and $s = ayb'xc \in A; (B; C)$. In a similar way we can prove $A; (B; C) \subseteq (A; B); C$.

Property 4. *";" distributes over arbitrary union in both arguments.*

Proof. Let I be a collection of subsets of T and $B \subseteq T$.

$$\begin{aligned}
& \bigcup(A : A \in I : A); B \\
= \quad & \{\text{definition ";"}\} \\
& \{a, x, b : ax \in \bigcup(A : A \in I : A) \wedge xb \in B : axb\} \cup inf.(\bigcup(A : A \in I : A)) \\
= \quad & \{\text{interchange unions, definition } inf\} \\
& \bigcup(A : A \in I : \{a, x, b : ax \in A \wedge xb \in B : axb\}) \cup \bigcup(A : A \in I : inf.A) \\
= \quad & \{\text{calculus}\} \\
& \bigcup(A : A \in I : \{a, x, b : ax \in A \wedge xb \in B : axb\} \cup inf.A) \\
= \quad & \{\text{definition}\} \\
& \bigcup(A : A \in I : A; B)
\end{aligned}$$

A similar proof can be given for the distributive property in the other argument.

Property 5. *";" is associative.*

Proof. Let $A, B, C \subseteq T$. We prove this property by showing $inf.((A; B); C) = inf.(A; (B; C))$ and $fin.((A; B); C) = fin.(A; (B; C))$.

$$\begin{aligned}
& inf.((A; B); C) \\
= \quad & \{\text{property } 2(i)\} \\
& inf.(A; B) \cup fin.(A; B); inf.C \\
= \quad & \{\text{property } 2(i), (ii)\} \\
& inf.A \cup (fin.A); inf.B \cup ((fin.A); fin.B); fin.C \\
= \quad & \{\text{property 3}\} \\
& inf.A \cup (fin.A); inf.B \cup (fin.A); ((fin.B); fin.C) \\
= \quad & \{\text{property 4}\} \\
& inf.A \cup (fin.A); (inf.B \cup (fin.B); fin.C)
\end{aligned}$$

$$
\begin{array}{ll}
= & \{\text{property } 2(i)\} \\
& \mathit{inf}.A \cup (\mathit{fin}.A); \mathit{inf}.(B;C) \\
= & \{\text{property } 2(i)\} \\
& \mathit{inf}.(A;(B;C))
\end{array}
$$

$$
\begin{array}{ll}
& \mathit{fin}.((A;B);C) \\
= & \{\text{property } 2(ii)\} \\
& (\mathit{fin}.(A;B));\mathit{fin}.C \\
= & \{\text{property } 2(ii)\} \\
& ((\mathit{fin}.A);\mathit{fin}.B);\mathit{fin}.C \\
= & \{\text{property } 3\} \\
& (\mathit{fin}.A);((\mathit{fin}.B);\mathit{fin}.C) \\
= & \{\text{property } 2(ii) \text{ twice (same steps)}\} \\
& \mathit{fin}.(A;(B;C))
\end{array}
$$

Property 6. *skip is a neutral element of ";" for both arguments.*

Proof. Immediate, from the definitions of ";" and *skip*.

Definition 7. *For $A \subseteq T$, we define the powers of A as follows.*

$$
\begin{aligned}
A^0 &= \mathit{skip} \\
A^{n+1} &= A; A^n \quad \text{for } n \geq 0.
\end{aligned}
$$

We need this notion in the definition of the repetition but we need more. As mentioned previously, in an execution of a repetition the body may be executed an infinite number of times. The operational interpretation is an infinite sequence as a limit of finite ones. We therefore continue with studying limits of sequences.

The prefix order on T is a partial order. We are going to define the above limit as a least upper bound of some set. Since T is not a complete lattice, not every set needs to have a least upper bound. However, we show that each ascending chain has one. Consider an ascending chain in T: $c_i, 0 \leq i$. There are two possibilities: there exists an $N \in I\!N$ such that $c_j = c_N$ for $j \geq N$ or no such N exists. In the first case we have $c_N = \sqcup\{i : 0 \leq i : c_i\}$. In the second case we can find a function $f : I\!N \rightarrow I\!N$ such that $c_{f.i}, 0 \leq i$ is strictly increasing. Define $d \in T$ elementwise as $d.i = c_{f.i}.i, 0 \leq i$. Notice that d is of infinite length. We show that d is the least upper bound of the chain. Suppose $c_k \not\sqsubseteq d$ for some k. Since $|d| = \infty$, this implies that there exists a j such that $c_k.j \neq d.j$. Hence, $c_k.j \neq c_{f.j}.j$. It follows that c_k and $c_{f.j}$ are incomparable which contradicts the assumption that the c_i form a chain. We conclude that d is an upper bound of the chain. Notice that every upper bound of $\{i : 0 \leq i : c_i\}$ is of infinite length. Assume e is another upper bound, different from d. We have that d and e differ at some index i. It follows that $c_{f.i}$ is not a prefix of e which is in contradiction with the assumption that e is an upper bound of the chain. We conclude that d is the least upper bound of the chain. It follows that, for

an ascending chain, the least upper bound is well defined. Notice however that T is not a cpo since the empty string is not an element of T.

Definition 8. *A characterizing chain for an infinite sequence $l \in T$ is an ascending chain $l_i, 0 \leq i$ such that*

$$(i)\ l_i \neq l_{i+1},$$
$$(ii)\ l = \sqcup\{i : 0 \leq i : l_i\}.$$

Definition 9. *l is a loop point of $A \subseteq T$ if there exists a characterizing chain $l_i, 0 \leq i$ for l such that $l_i \in A^i$. The set of loop points of A is denoted by loop.A.*

Property 10. *For $A \subseteq T$ we have $loop.A \cup inf.A = A; loop.A$.*

Proof. The proof is by mutual inclusion. First we prove $\subseteq$. Clearly, $inf.A \subseteq A; loop.A$. Choose $l \in loop.A$ and $l_i, 0 \leq i$ a characterizing chain for l. From the definition of characterizing chain, it follows that all l_i are finite. We can write $l_i, 1 \leq i$ as $l_1 m_i$ since the l_i form an ascending chain and $l = l_1 m$ since l is the least upper bound of the l_i. We now have $(last.l_1)m_i \in \{last.l_1\}; A^{i-1}$. We construct a new chain, $n_i, 0 \leq i$, as follows: $n_i = (last.l_1)m_{i+1}$. The limit of this chain is $(last.l_1)m \in loop.A$. Since $l_1 \in A$, $\{l_1\}; (last.l_1)m \in A; loop.A$.

For the other inclusion, choose $l \in A; loop.A$. Then either $l = axm$ for $ax \in fin.A, xm \in loop.A$ or $l \in inf.A$. In the last case we are done. In the first case, we have a characterizing chain for xm, $xm_i, 0 \leq i$. We construct a new chain, l_i, as follows. $l_0 = (ax).0$, $l_i = axm_{i-1}, 1 \leq i$. Now we have $l_i \in A^i, 0 \leq i$ and $l = \sqcup\{i : 0 \leq i : l_i\}$ hence $l \in loop.A$.

Definition 11. *For $A \subseteq T$ we define A^ω as follows.*

$$A^\omega = \bigsqcup(n : 0 \leq n : A^n) \cup loop.A$$

Now we are ready to give the definition of the repetition. We abbreviate **do** $B \rightarrow S$ **od** by DO.

$$DO = (B^{guard}; S)^\omega; \neg B^{guard}$$

We have to verify that we defined a proper element of *Prog*. Suppose there exists $x \in X$ such that there is no sequence in DO starting with x. Since sequences starting with x are present in $(B^{guard}; S)^\omega$, it follows that they are all finite and terminate in a state satisfying B. Define a characterizing chain as follows. $l_0 = x$, choose $l_{i+1} \in \{l_i\}; S$, arbitrarily. The least upper bound of this chain is a loop point of $(B^{guard}; S)$ starting with x. Since it is a loop point, it is in DO. This is a contradiction. It follows that DO is an element of *Prog*.

The above definition of DO is a rather complicated one. It has the advantage however, of defining DO uniquely. A more conventional definition of DO starts with the first unfolding. A definition of DO is then obtained by solving

$$DO = \textbf{if } B \rightarrow S; DO \,[]\, \neg B \rightarrow skip \textbf{ fi}$$

viewed as an equation in sets. This equation itself however, cannot serve as a definition because it does not have a unique solution. This means that we must distinguish among the solutions, which may require a topological approach. This is done for instance in [8] where it is proven that only one solution of the above equation exists that is nonempty and closed with respect to a certain topology, based on a metric. This proof is given only for a language with bounded choice. In our definition we could restrict ourselves to the prefix order on strings and we did not need a metric or a topology. Furthermore, we do not need any restrictions on the alternative construct.

The main reason that we deviate from the usual definition in terms of the first unfolding is that this semantics serves as the basis of a generalized notion of weakest precondition. This generalization allows us to introduce weakest preconditions for various properties. We use the first unfolding to obtain a recursive equation for the weakest precondition of a repetition. We use the above definition to distinguish among the solutions of this equation.

Since we are going to need that DO is semantically equivalent to its first unfolding, we prove it as a theorem. This is done in the remainder of this section.

Lemma 12. $S^\omega = \mathit{skip} \cup S; S^\omega$

Proof.

$$
\begin{array}{ll}
 & S^\omega \\
= & \{\text{definition}\} \\
 & \bigcup(n : 0 \leq n : S^n) \cup \mathit{loop}.S \\
= & \{\text{property 10: } \mathit{inf}.S \subseteq \bigcup(n : 0 \leq n : S^n)\} \\
 & \bigcup(n : 0 \leq n : S^n) \cup S; \mathit{loop}.S \\
= & \{\text{definition 7, domain split}\} \\
 & \mathit{skip} \cup \bigcup(n : 1 \leq n : S^n) \cup S; \mathit{loop}.S \\
= & \{\text{definition 7}\} \\
 & \mathit{skip} \cup \bigcup(n : 0 \leq n : S; S^n) \cup S; \mathit{loop}.S \\
= & \{\text{property 4}\} \\
 & \mathit{skip} \cup S; \bigcup(n : 0 \leq n : S^n) \cup S; \mathit{loop}.S \\
= & \{\text{property 4}\} \\
 & \mathit{skip} \cup S; (\bigcup(n : 0 \leq n : S^n) \cup \mathit{loop}.S) \\
= & \{\text{definition 11}\} \\
 & \mathit{skip} \cup S; S^\omega
\end{array}
$$

Theorem 13. $DO = \textbf{if } B \rightarrow S; DO \; [] \neg B \rightarrow \mathit{skip} \textbf{ fi}$

Proof.

$$
\begin{aligned}
& \textbf{if } B \to S; DO \,[]\neg B \to skip \textbf{ fi} \\
= \quad & \{\text{definition}\} \\
& B^{guard}; S; DO \cup \neg B^{guard}; skip \cup false^{guard}; abort \\
= \quad & \{false^{guard} = \emptyset, \text{ definition } DO\} \\
& B^{guard}; S; (B^{guard}; S)^{\omega}; \neg B^{guard} \cup \neg B^{guard}; skip \\
= \quad & \{\text{property 6, twice}\} \\
& B^{guard}; S; (B^{guard}; S)^{\omega}; \neg B^{guard} \cup skip; \neg B^{guard} \\
= \quad & \{\text{property 4}\} \\
& (B^{guard}; S; (B^{guard}; S)^{\omega} \cup skip); \neg B^{guard} \\
= \quad & \{\text{lemma 12}\} \\
& (B^{guard}; S)^{\omega}; \neg B^{guard} \\
= \quad & \{\text{definition}\} \\
& DO
\end{aligned}
$$

This concludes our definition of the operational semantics of the guarded command language.

3 Properties of Programs

In [6], attention is focused on one particular property of a program, viz., whether or not it terminates in a state that satisfies a certain condition. It has been recognized that, especially in the area of parallel programming, other properties are important as well. For instance, the properties "during execution, a condition will become true" and "a state satisfying a certain condition is always followed by a state satisfying some other condition" are properties defined and used in temporal logic. The fact that a program has a certain property means that all its computations share a common characteristic. In other words, a property may be regarded as a set of computations and a program has a property if all its computations are in the property. We add to our notation

$$Prop = \mathcal{P}(T), \text{ the set of properties.}$$

For a particular program and property it is fairly well possible that only part of the computations of the program are in the property. The function $wp.S.p$ defines the *weakest precondition* for a program S such that all its computations (starting from that precondition) have the property: termination in a state satisfying p (and similarly, wlp). This is generalized to other properties as follows.

Definition 14. *The weakest precondition is a function* $w : Prog \times Prop \to P$.

$$w.S.Q.x \;=\; \{x\}; S \subseteq Q$$

Now we introduce wlp and wp as special cases.

Definition 15. *The termination functions* $lt, t : P \to Prop$ *are defined by*

$$lt.p = \{s, w : sw \in T \wedge w \in X \wedge (|sw| = \infty \vee p.w) : sw\},$$
$$t.p = \{s, w : sw \in T \wedge w \in X \wedge |sw| < \infty \wedge p.w : sw\}.$$

The functions $wlp, wp : Prog \times P \to P$ *are defined by*

$$wlp.S.p = w.S.(lt.p),$$
$$wp.S.p = w.S.(t.p).$$

We use the operational semantics to derive *wlp* and *wp* for the constructs in the language. We do not derive them all; some of them are left to the reader. From now on we will often use a somewhat different characterization of sequential composition. From property 4 it follows that for $S, U \subseteq T$,

$$S;U = \bigcup(s, x : sx \in S \wedge x \in X : \{sx\};U).$$

For arbitrary program S we have

$$\begin{array}{ll} & wlp.S.p.x \\ = & \{\text{definition } wlp\} \\ & w.S.(lt.p).x \\ = & \{\text{definition } w\} \\ & \{x\};S \subseteq lt.p \end{array}$$

and a similar formula for *wp*. This yields for *skip*

$$\begin{array}{ll} & wlp.skip.p.x \\ = & \{\text{definition } skip\} \\ & \{x\};X \subseteq lt.p \\ = & \{\text{definition ";"}\} \\ & \{x\} \subseteq lt.p \\ = & \{\text{definition } lt\} \\ & p.x. \end{array}$$

For $y := e$,

$$\begin{array}{ll} & wlp.(y := e).p.x \\ = & \{\text{definitions assignment and ";"}\} \\ & \{x(x[y/e.x])\} \subseteq lt.p \\ = & \{\text{definition } lt\} \\ & p.(x[y/e.x]). \end{array}$$

For both the assignment and *skip*, *wlp* and *wp* coincide. In a similar way we derive

$$wlp.abort.p = true,$$
$$wp.abort.p = false.$$

Sequential composition is a little more complicated.

$$
\begin{aligned}
& wlp.(S;U).p.x \\
= \quad & \{\text{definition } wlp\} \\
& \{x\};S;U \subseteq lt.p \\
= \quad & \{\text{definitions ";" and } lt\} \\
& \forall(t,w : tw \in \bigcup(s,v : sv \in \{x\};S \wedge v \in X : \{sv\};U) \wedge w \in X : \\
& \qquad |tw| = \infty \vee p.w) \\
= \quad & \{tw \in \bigcup(s,v : sv \in \{x\};S \wedge v \in X : \{sv\};U) \equiv \\
& \qquad tw = svuw \text{ for } sv \in \{x\};S, vuw \in \{v\};U\} \\
& \forall(s,v,u,w : sv \in \{x\};S \wedge v \in X \wedge vuw \in \{v\};U \wedge w \in X : \\
& \qquad |svuw| = \infty \vee p.w) \\
= \quad & \{\text{calculus}\} \\
& \forall(s,v,u,w : sv \in \{x\};S \wedge v \in X \wedge vuw \in \{v\};U \wedge w \in X : |sv| = \infty \vee \\
& \qquad |vuw| = \infty \vee p.w) \\
= \quad & \{\text{nesting}\} \\
& \forall(s,v : sv \in \{x\};S \wedge v \in X : \forall(u,w : vuw \in \{v\};U \wedge w \in X : |sv| = \infty \vee \\
& \qquad |vuw| = \infty \vee p.w)) \\
= \quad & \{u \text{ and } w \text{ not free in } sv, \text{ renaming the dummy}\} \\
& \forall(s,v : sv \in \{x\};S \wedge v \in X : |sv| = \infty \vee \\
& \qquad \forall(u,w : uw \in \{v\};U \wedge w \in X : |uw| = \infty \vee p.w)) \\
= \quad & \{\text{definition } lt\} \\
& \forall(s,v : sv \in \{x\};S \wedge v \in X : |sv| = \infty \vee \{v\};U \subseteq lt.p) \\
= \quad & \{\text{definition } wlp\} \\
& \forall(s,v : sv \in \{x\};S \wedge v \in X : |sv| = \infty \vee wlp.U.p.v) \\
= \quad & \{\text{definition } wlp\} \\
& wlp.S.(wlp.U.p).x
\end{aligned}
$$

In a similar way we obtain $wp.(S;U).p = wp.S.(wp.U.p)$. The alternative construct is simple again and yields

$$
\begin{aligned}
& wlp.IF.p = \forall(i : B_i : wlp.S_i.p), \\
& wp.IF.p = \exists(i :: B_i) \wedge \forall(i : B_i : wp.S_i.p).
\end{aligned}
$$

The repetition is the most interesting construct. Using theorem 13 we derive

$$
\begin{aligned}
& wlp.DO.p.x \\
= \quad & \{\text{theorem 13}\} \\
& wlp.(\textbf{if } B \rightarrow S; DO \; [] \neg B \rightarrow \textit{skip} \textbf{ fi}).p.x \\
= \quad & \{wlp \text{ for } IF, \text{";" and } \textit{skip}\} \\
& (B \wedge wlp.S.p.(wlp.DO.p) \;\vee\; \neg B \wedge p).x.
\end{aligned}
$$

We conclude that $wlp.DO.p$ is a solution of the equation in unknown predicate Y:

$$[Y \equiv B \wedge wlp.S.Y \vee \neg B \wedge p] \tag{1}$$

According to the theorem of Knaster-Tarski ([5, 6]) this equation has extreme solutions if the righthandside is a monotonic function of Y, i.e.,

$$[Y \Rightarrow Y'] \Rightarrow [(B \wedge wlp.S.Y \vee \neg B \wedge p) \Rightarrow (B \wedge wlp.S.Y' \vee \neg B \wedge p)]$$

which reduces to

$$[Y \Rightarrow Y'] \Rightarrow [wlp.S.Y \Rightarrow wlp.S.Y'].$$

This is a direct consequence of the definition of wlp. Hence, equation (1) has a weakest and a strongest solution.

Theorem 16. *$wlp.DO.p$ is the weakest solution of (1).*

Proof. Let y be an arbitrary solution of (1). We have to prove that y implies $wlp.DO.p$. Choose x such that $y.x$ holds and $s \in \{x\}; DO$. It is our obligation to prove that $wlp.S.p.x$ holds, i.e.,

$$|s| < \infty \Rightarrow p.(last.s)$$

We first prove by induction on n:

$$|t| < \infty \wedge t \in \{x\}; (B^{guard}; S)^n \Rightarrow y.(last.t) \tag{2}$$

For $n = 0$ we have the case $t = x$ and $y.x$ is given. Consider a finite sequence $t \in \{x\}; (B^{guard}; S)^{n+1}$. Then $t = avb, av \in \{x\}; (B^{guard}; S)^n, v \in X$. According to the induction hypothesis, $y.v$ holds. Since $vb \in \{x\}; B^{guard}; S, B.v$ holds. From the fact that y solves (1) we conclude $wlp.S.y.v$. This yields $y.(last.b)$, hence $y.(last.t)$ which concludes our proof of (2).

Assuming $|s| < \infty$ we can write s as tww with $tw \in \{x\}; (B^{guard}; S)^n$ for some $n \in \mathbb{N}$ and $w \in X$ (the doubling of the last element stems from the $\neg B^{guard}$ in the definition of DO). Now we have

$$\begin{array}{ll} & tww \in DO \\ \Rightarrow & \{(2), \text{definition } DO\text{: finite sequences end in } \neg B\} \\ & \neg B.w \wedge y.w \\ \Rightarrow & \{y \text{ is a solution of (1)}\} \\ & p.w \end{array}$$

which is what we had to prove.

Using theorem 13 again we obtain a similar equation for wp. It follows that $wp.DO.p$ is a solution of the equation in unknown predicate Y:

$$[Y \equiv B \wedge wp.S.Y \vee \neg B \wedge p] \tag{3}$$

Also $wp.S$ is a monotonic function. We have

Theorem 17. *wp.DO.p is the strongest solution of (3).*

Proof. We have to prove $[wp.DO.p \Rightarrow y]$ for every solution y of (3). We prove this by contradiction. Let y be such a solution and suppose there exists $x \in X$ such that $wp.DO.p.x \wedge \neg y.x$. We show the existence of a characterizing chain l_k, $k \geq 0$ starting in x of a loop point l such that $\forall(k : 0 \leq k : \neg y.(last.l_k))$. We have to show that we can find such an l_k in every $\{x\};(B^{guard};S)^k$. We prove this by induction. For $k = 0$ we have $l_0 = x$ and $\neg y.x$ is given. Suppose we have $l_k \in \{x\};(B^{guard};S)^k$ satisfying $l_k < \infty$ (any infinite computation starting in x is already a contradiction). We denote the last element of l_k by w and prove that $B.w$ holds.

$$
\begin{array}{ll}
 & \neg B.w \\
= & \quad \{\text{definition } DO\} \\
 & l_k w \in \{x\};DO \wedge \neg B.w \\
= & \quad \{wp.DO.p.x \text{ is given, } l_k \text{ is finite}\} \\
 & l_k w \in t.p \wedge \neg B.w \\
= & \quad \{\text{definition } t\} \\
 & p.w \wedge \neg B.w \\
= & \quad \{y \text{ solves (3)}\} \\
 & y.w \\
= & \quad \{\text{induction hypothesis: } \neg y.w\} \\
 & false
\end{array}
$$

We now have $B.w \wedge \neg y.w$. In view of y being a solution of (3) this implies $\neg wp.S.y.w$ hence there exists $s \in \{l_k\};B^{guard};S$ such that $s \notin t.y$. Because $wp.DO.p.x$ holds, $|s| < \infty$. This s is our l_{k+1}.

The limit of this chain is a loop point starting in x which is in DO. Since a loop point is infinite, this contradicts the assumption $wp.DO.p.x$.

This concludes our discussion of *wlp* and *wp*.

We proceed with some comments on the notion of nondeterminate programs. In [6], the following definition is given: program S is deterministic if

$$[wp.S.p \equiv \neg wlp.S.\neg p], \text{ for all } p. \tag{4}$$

We change the terminology a bit; a program that satisfies (4) is called *determinate* since there may be more than one computation but with respect to the final states these computations are indistinguishable. A program is deterministic if there is only one possible computation for every initial state, i.e., $|\{x\};S| = 1$ for every $x \in X$.

As a definition of determinate programs, (4) does not suffice anymore since we generalized the notion of weakest precondition. As an example, according to this definition the following program is determinate.

if $true \rightarrow x := 3; y := 4$
[] $true \rightarrow y := 4; x := 3$
fi

As long as we are interested in final states only, the two possibilities in this program are indistinguishable. If we generalize this idea to other properties we can say that a program is determinate if the fact whether a computation is in a property is completely determined by the initial state. In other words, there are no initial states with computations both inside and outside the property.

Definition 18. *Program S is determinate with respect to property $Q \in Prop$ if*

$$\{x\};S \subseteq Q \ \vee \ \{x\};S \subseteq (T \setminus Q)$$

or equivalently,

$$\neg(\{x\};S \subseteq Q) \ \equiv \ \{x\};S \subseteq (T \setminus Q)$$

for all $x \in X$. S is determinate with respect to a class of properties if it is determinate with respect to all properties in the class.

Using the definition of weakest precondition we derive that program S is determinate with respect to property Q if

$$[\neg w.S.Q \ \equiv \ w.S.(T \setminus Q)].$$

Now we analyze what this means for wp and wlp. If we look at one particular postcondition p we obtain for wp:

$$\begin{array}{ll} & T \setminus t.p \\ = & \quad \{\text{definition } t\} \\ & \{t : t \in T \wedge (|t| = \infty \vee \neg p.(last.t)) : t\} \\ = & \quad \{\text{definition } lt\} \\ & lt.\neg p \end{array}$$

Hence, program S is determinate with respect to termination in postcondition p if

$$[\neg wp.S.p \ \equiv \ wlp.S.\neg p].$$

By quantification over all postconditions we obtain the original definition.

Notice that we are quite fortunate to be able to characterize the complement of $t.p$ (and, of course, of $lt.p$). For other properties we may not find such a concise characterization.

Finally, we have a look at some other properties of programs. In recent work weakest preconditions were introduced to reason about alternative properties of programs ([11, 7, 12]) in an axiomatic way. The above operational semantics can be used for properties like these as well. (As a matter of fact, the introduction of these properties was the major motivation to start this work in the first place.) Consider for example the property: "ever, during execution, condition q holds" (the operator "$\diamond$" from temporal logic). This property is captured in

$$ever.q = \{s, w, t : swt \in T \wedge |s| < \infty \wedge w \in X \wedge q.w : swt\}.$$

We are interested in determining the weakest precondition for a program such that the computations, starting in a state satisfying this precondition are in *ever.q*. Hence we are interested in the function *weakest ever*.

$$wev.S.q = w.S.(ever.q)$$

Again we can analyse what this yields for the various language constructs. As it turns out, sequential composition is a problem. Informally, this is seen as follows: a computation in $S;U$ that is in $ever.q$ may reach a state satisfying q either during S or during U. But

$$wev.(S;U).q = wev.S.q \vee wp.S.(wev.U.q)$$

is not the right formula since it does not allow initial states in which computations of both flavors start. We have to include, therefore, the final state in the definition of the function. As a consequence, we also have a liberal version.

$$wev.S.q.r = w.S.(ever.q \cup t.r)$$
$$wlev.S.q.r = w.S.(ever.q \cup lt.r)$$

By a whole lot of string manipulation we obtain now

$$wev.(S;U).q.r = wev.S.q.(wev.U.q.r)$$
$$wlev.(S;U).q.r = wlev.S.q.(wlev.U.q.r).$$

The fact that the final state of a program is of interest, even if the property that we are interested in does not refer explicitly to the final state comes from the fact that we look at preconditions, combined with the existence of sequential composition in our programming language. As a result, this final state will show up in any function that we introduce. This is quite unfortunate since it increases the number of arguments that our functions have. Consider as a second example the property *leads-to.p.q* defined as follows.

$$leads\text{-}to.p.q \;=\; \{s : \forall(a,b : s = ab \wedge |a| < \infty : b \in ever.p \Rightarrow b \in ever.q) : s\}$$

Informally, a computation is in *leads-to.p.q* if a state satisfying p is always followed by a state satisfying q (where "followed by" is understood to include coincidence). In the function *wto*, we include the final state for the same reason as we did it for *wev*.

$$wto.S.p.q.r = w.S.(leads\text{-}to.p.q \cup t.r)$$

For sequential composition, the following function is obtained.

$$wto.(S;U).p.q.r = wto.S.p.q.(wev.U.q.r) \wedge wlp.S.(wto.U.p.q.r)$$

In [9], these properties are analyzed in detail. Especially interesting are the results for the repetition. Using theorem 13, recursive equations are obtained for both properties. The function $wev.DO.q.r$ turns out to be the strongest solution of its equation

while $wto.DO.p.q.r$ is the weakest solution. This is especially remarkable since both of them are true generalizations of wp.

$$wev.S.false.r = wp.S.r$$
$$wto.S.true.false.r = wp.S.r$$

Reasoning about programs in an operational domain is quite cumbersome. Therefore, it is a good idea to isolate some characteristics of the properties of interest and to use these characteristics, together with an axiomatic definition of the properties to prove facts about programs. This is done in [12, 9].

4 Conclusion

In this paper we have presented an operational semantics for the guarded command language in terms of sequences of states. The definition of the repetition was given in a closed form which allowed us to derive that the repetition is equivalent to its first unfolding. In this way we could prove that $wp.DO.p$ is the strongest solution of its defining equation and $wlp.DO.p$ the weakest. A generalization of the notion of weakest precondition allows us to reason about different properties in a similar way as we do about final states. Although we did not exploit that in detail in this paper we gave examples of two temporal properties. The generalization also gave rise to a nice characterization of nondeterminate programs.

Originally, this work started from our interest in parallel programs. We did not address any parallelism at all in this paper. However, we did describe nondeterministic choice. This means that we can describe parallelism in a similar way as, for instance, in Unity ([4], apart from the fairness constraint) or in action systems ([2, 3]), viz., as the interleaving of atomic actions.

5 Acknowledgements

I want to thank the two referees for helpful comments on the first version of the paper. The major part of this paper was part of my thesis and I want to thank my supervisor Jan van de Snepscheut for discussing the subjects presented in this paper and for his support. I also thank Wim Hesselink for his comments on notation and presentation.

References

1. Alpern, B., Schneider, F.B.: Defining liveness. I.P.L. **21** (1985) 181-185
2. Back, R.J.R.: Refinement calculus, part II: parallel and reactive programs. report ser. A, no. 93 (1989) Åbo Akademi Finland
3. Back R.J.R., Sere, K.: Stepwise refinement of action systems. In: Mathematics of program construction (J.L.A. van de Snepscheut (ed)), Springer-Verlag LNCS **375** (1989) 115-138
4. Chandy, K.M., Misra, J.: Parallel programming, a foundation. Addison-Wesley publishing company, Reading 1988

5. Davey, B.A., Priestley, H.A.: Introduction to lattices and order. Cambridge University Press, Cambridge 1990
6. Dijkstra, E.W., Scholten, C.S.: Predicate calculus and program semantics. Springer-Verlag, New York 1990.
7. Knapp, E., A predicate transformer for progress, I.P.L. **33** (1989/1990) 323-330
8. Kuiper, R.: An operational semantics for bounded nondeterminism equivalent to a denotational one. In: Algorithmic Languages, de Bakker/van Vliet (eds) 373-398, IFIP, North Holland 1981
9. Lukkien, J.J.: Parallel program design and generalized weakest preconditions. PhD thesis, Groningen University The Netherlands 1991
10. Manna, Z., Pnueli, A.: How to cook a temporal proof system for your pet language. Proceedings POPL (ACM) 141-153, Austin 1983
11. Morris, J.M.: Temporal predicate transformers and fair termination. Acta Informatica **27** (1990) 287-313
12. van de Snepscheut, J.L.A., Lukkien, J.J.: Weakest preconditions for progress. Formal Aspects of Computing **4** (1992) 195-236