Ralph-Johan Back | Viorel Preoteasa

Semantics and Proof Rules of Invariant Based Programs

Turku Centre *for* Computer Science

TUCS Technical Report
No 903, June 2008

Semantics and Proof Rules of Invariant Based Programs

Ralph-Johan Back
Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3-5 B, 20520 Turku, Finland

Viorel Preoteasa
Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3-5 B, 20520 Turku, Finland

TUCS Technical Report
No 903, June 2008

Abstract

Invariant based programming is an approach where we start to construct a program by first identifying the basic situations (pre- and postconditions as well as invariants) that could arise during the execution of the algorithm. These situations are identified before any code is written. After that, we identify the transitions between the situations, which will give us the flow of control in the program. The transitions are verified at the time when they are constructed. The correctness of the program is thus established as part of constructing the program. The program structure in invariant based programs is determined by the information content of the situations, using *nested invariant diagrams.* The control structure is secondary to the situation structure, and will usually not be well-structured in the classical sense, i.e., it is not necessarily built out of single-entry single-exit program constructs.

The execution of an invariant diagram may start in any situation and will choose one of the enabled transitions in this situation, to continue to the next situation. In this way, the execution proceeds from situation to situation. Execution terminates when a situation is reached from which there are no enabled transitions. Because the execution could start and terminate in any situation, invariant-based programs can be thought of as multiple entry, multiple exit programs. The transitions may have statements with unbounded nondeterminism, because we allow specification statements in transitions. Invariant based programs are thus a considerable generalization of ordinary structured program statements, and defining their semantics and proof theory provides a challenge that usually does not arise for more traditional programming languages

We study in this paper the semantics and proof rules for invariant-based programs. The total correctness of an invariant diagram is established by proving that each transition preserves the invariants and decreases a global variant. The proof rules for invariant-based programs are shown to be correct and complete with respect to an operational semantics. The proof of correctness and completeness introduces the weakest precondition semantics for invariant diagrams, and uses a special construction, based on well-ordered sets, of the least fixpoint of a monotonic function on a complete lattice. The results presented in this paper have been mechanically verified in the PVS theorem prover.

TUCS Laboratory
Software Construction Laboratory

1 Introduction

Invariant based programming is an approach where we start to construct a program by first identifying the basic situations (pre- and postconditions as well as invariants) that could arise during the execution of the algorithm. These situations are identified before any code is written. After that, we identify the transitions between the situations, which will give us the flow of control in the program. The transitions are verified at the time when they are constructed. The correctness of the program is thus established as part of constructing the program. The program structure in invariant based programs is determined by the information content of the situations, using *nested invariant diagrams*. The control structure is secondary to the situation structure, and will usually not be well-structured in the classical sense, i.e., it is not necessarily built out of single- entry single-exit program constructs. We refer to a program constructed in this manner as an *invariant based program*.

The execution of an invariant based program may start in any situation and will choose one of the enabled transitions in this situation, to continue to the next situation. In this way, the execution proceeds from situation to situation. Execution terminates when a situation is reached from which there are no enabled transitions. Because the execution could start and terminate in any situation, invariant-based programs can be thought of as multiple entry, multiple exit programs. Termination of a program may also happen anywhere, not just at some prespecified exit points. The transitions may have statements with unbounded nondeterminism, because we allow specification statements in transitions. Invariant based programs are thus a considerable generalization of ordinary structured program statements, and defining their semantics and proof theory provides a challenge that usually does not arise for more traditional programming languages

We study here the semantics and proof theory of invariant based programs [3, 4, 5]. The idea of invariant based programming is not new, similar ideas were proposed in the 70's by John Reynolds [16], Martin van Emden [18], and Ralph-Johan Back [3, 4], in different forms and variations. Dijkstra's later work on program construction also points in this direction [9], where he emphasizes the formulation of a loop invariant as a central step in deriving the program code. However, Dijkstra insists on building the program in terms of well-structured (single-entry single-exit) control structures, whereas there are no restrictions on the control structure in invariant based programming. Basic for these approaches is that the loop invariants are formulated before the program code is written. Eric Hehner [10] was working along similar lines, but chose relations rather than predicates as the basic construct.

Invariant based programs are intended to be correct by construction, so proof of correctness is part of the programming process. For that purpose, we need to define the semantics of invariant based programs, give proof rules for showing that the program is correct, and we need to show that these proof rules are sound

(and preferably complete). But we cannot use existing theories directly, as they are typically based on well-structured control constructs. Our purpose here is therefore to define the semantics and proof theory of invariant based programs from scratch, and to show that the proof rules we give are both sound and complete with respect to the semantics we give for invariant based programs.

We will proceed in the following way. We first describe invariant based programs in an intuitive way, to give a feel for the basic ideas behind this approach, and for the constraints and generalizations inherent in this approach. We begin the theoretical study of invariant based programs by defining their operational semantics. We will in fact define two different operational semantics. The first one is a small-step operational semantics that describes the way in which an invariant based program is executed by a computer. In essence, the small step semantics describes an interpreter for the programming language. The second one is a big step operational semantics that describes the overall behavior of an invariant based program, essentially as a mapping from input states to possible output states. This semantics is only concerned with the input output behavior of the program. It allows us to define basic properties of program execution, like partial correctness and termination. We show that the small step semantics and the big step semantics are equivalent. In other words, any correctness property that holds for the big step semantics of an invariant based program will also hold for the small step execution of the program, and vice versa.

We then define a weakest precondition semantics for invariant based programs. The weakest precondition semantics is compositional, and allows us to directly compute the basic correctness properties of an invariant based program. We show that the weakest precondition semantics is equivalent to the big step operational semantics.

The weakest precondition semantics does not, however, give us a practical method for proving program correctness, because it uses least fixpoints to determine the semantics of loops. We get around this obstacle by giving a collection of Hoare-like [11] total correctness proof rules for invariant based program. We show that the proof rules are sound with respect to the weakest precondition semantics. This means that if we prove, using these proof rules, that our invariant based program is correct, then it will also be correct according to the weakest precondition semantics.

Because we have shown that the weakest precondition semantics is equivalent to the big step semantics, which in turn is equivalent to the small step semantics, we get the following basic property: If we have proved that an invariant based program is correct using the given proof rules, then any execution of the invariant based program that respects the small step semantics will be correct. This means that our proof system is *sound*.

We also study the converse problem: Assume that we have a correct invariant based program that is executed according to the small step semantics. Can we then prove that the program is correct using the given proof rules for invariant

based programs? The answer to this question is positive, i.e., our proof system is also *complete*. In the end, this means that our proof system is both sound and complete for invariant based programs.

The theory of invariant based programs has been completely mechanized in the PVS interactive proof system [15]. This gives a very solid foundation for our results. This PVS formalization depends on the well-ordering theorem which says that any set can be well-ordered.

Both the soundness and completeness results we have for invariant based programs are consequences of more general results for monotonic functions on a complete lattice. We give a special construction, based on a well ordered set, of the least fixpoint of a monotonic function on a complete lattice. The completeness theorem is a consequence of this construction. We allow specification statements in our programs, so our semantics may have unbounded nondeterminism. This means that we need to go beyond natural numbers and use well ordered relations [13] or ordinals [1] when proving completeness. This is due to the fact that unbounded non-deterministic statements are not continuous. Nipkow [13] presents an Isabelle [14] formalization of complete Hoare proof rules for recursive parameterless procedures in the context of unbounded nondeterminism. Our programming language is, however, more general than the one studied in [13], because it features multiple-entry, multiple-exit statements, and a more general recursion mechanism. Our proof of completeness is also more general and simpler than the one in [13], and we believe that it could be applied unmodified to richer programming constructs, such as procedures with parameters and local variables.

The framework that we build allows us to study termination proofs for invariant based programs in more detail, and to justify specific proof rules for termination. Proving termination of invariant based programs is more difficult than usually, because the control structure is unconstrained. This means that our loops need not be nested, we can have intersecting loops, loops with exits in the middle (multiple exit loops) and loops that can be entered in the middle (multiple entry loops). Nevertheless, the framework that we have built allows us to formulate new proof techniques for termination, that are more general than existing ones, but which we still can prove to be sound.

2 Syntax of invariant diagrams

Let State be an unspecified type of *states* and Var be the type of all program variables. For $x \in \mathsf{Var}$, the type of the variables x, denoted $\mathsf{T}.x$, contains all values that can be assigned to x. Intuitively a state s from State gives the values to the program variables. Formally, we access and update program variables using two functions. $\mathsf{val}.x : \mathsf{State} \to \mathsf{T}.x$ and $\mathsf{set}.x : \mathsf{T}.x \to \mathsf{State} \to \mathsf{State}$. For $x \in \mathsf{Var}$, $s \in \mathsf{State}$, and $a \in \mathsf{T}.x$, $\mathsf{val}x.s$ is the value of x in state s, and $\mathsf{set}x.a.s$ is the

state obtained from s by setting the value of location x to a. The behavior of these functions is described using a set of axioms [6]. For the purpose of this paper we do not need to consider in greater details the treatment of program variables.

Let Bool be the set of Boolean values. Predicates, denoted Pred, are the functions from State $\rightarrow$ Bool. Relations, denoted by Rel, are functions from State to Pred. We denote by $\subseteq$, $\cup$, and $\cap$ the predicate inclusion, union, and intersection respectively. The type Pred together with inclusion forms a complete Boolean algebra.

We use higher-order logic [7] as the underlying logic. If $f : A \rightarrow B$ is a function and $x \in A$, then the function application is denoted by $f.x$ (f dot x).

An *invariant diagram* is a directed graph where nodes are labeled with *invariants* (predicates) and edges are labeled with *transitions* (*program statements*). The transitions are non-iterative programs built from assertions, assumptions, demonic updates, demonic choices, and sequential compositions. The abstract syntax of transitions is defined by the following recursive data type:

$$\begin{array}{rcl} \mathsf{Trs} & = & Assert(\mathsf{Pred}) \\ & & \mid Assume(\mathsf{Pred}) \\ & & \mid Update(\mathsf{Rel}) \\ & & \mid Choice(\mathsf{Trs},\ \mathsf{Trs}) \\ & & \mid Comp(\mathsf{Trs},\ \mathsf{Trs}) \end{array}$$

If p is a predicate, R is a relation, and $S,\ T$ are transitions, then we use the notations $\{p\},\ [p],\ [R],\ S \sqcap T,\ S\,;\,T$ for the constructs $Assert$, $Assume$, $Update$, $Choice$, and $Comp$, respectively. Intuitively the execution of the *assert* statement $\{p\}$ and the *assume* statement $[p]$ starting in a state s in which p is true behave as *skip*. If p is false in s, then $\{p\}$ *fails* and $[p]$ is not *enabled*. The *demonic update* $[R]$, when starting in a state s, terminates in a nondeterministically chosen state s' such that $R.s.s'$. If there is no state s' such that $R.s.s'$, then $[R]$ is not enabled. The execution of the *demonic choice* $S \sqcap T$ nondeterministically chooses S or T. The transition $S\,;\,T$ is the *sequential composition* of the transitions S and T.

We model both *assignments* and *nondeterministic assignments* using the demonic update:

$$(x := e) \quad = \quad [\lambda s, s' \bullet s' = \mathsf{set}.x.(e.s).s]$$

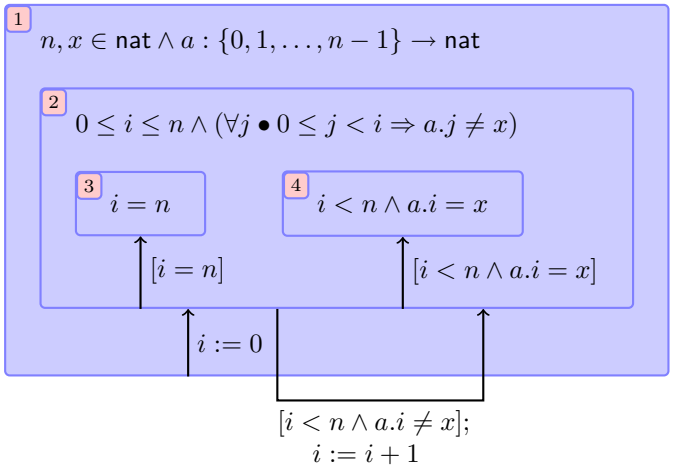
$$[x := a \bullet b.a] \quad = \quad [\lambda s, s' \bullet (\exists a \bullet s' = \mathsf{set}.x.a.s \wedge b.a.s)]$$

A transition S is *enabled*, when starting from a state s, if it is possible to avoid any assume or demonic choice statements which are not enabled. For example the transition $S = ([x < 4]; x := x + 1; [x > 1]) \sqcap ([x > 10]; x := 3)$ is enabled for all states where x is $1,\ 2,\ 3$ or greater than 10. If x is 1 in the initial state s, then we chose the first part of the choice in S, and all assume statements in this part are enabled. The *guard* of a transition S is a predicate which is true for all states from which S is enabled. We will define formally later the notions enabled

and guard, but we have introduced them here informally to explain the intuition behind invariant based programs.

Let I be a nonempty set of indexes. Formally an *invariant diagram* ID is a tuple (P, D) where $P : I \to \mathsf{Pred}$ are the *invariants* and $D : I \times I \to \mathsf{Trs}$ are the *transitions*. D is called a *transition diagram* and the elements of I are called *situations*. The invariant diagrams are represented as special graphs. The nodes are represented by rectangles. Inside the rectangles we write the invariants. The transitions are represented by directed edges in the graph, labeled with the transition statements.

Figure 1 represents an invariant diagram. The program represented in this figure searches if an element is member in an array of numbers.



(1)

In Figure 1 the situations are 1, 2, 3, 4. In practice, it is very often the case that the invariant of a situation i is stronger than the invariant of another situation j ($P_i = P_j \land q$). In this case we draw the situation i inside situation j, and we label i only with the predicate q. The invariant of situation i is the conjunction of q and the labels of all situations containing the situation i. For example in (1), the invariant of situation 4 is the conjunction of the predicate labels of situations 1, 2, and 4:

$$(n, x \in \mathsf{nat} \land a : \{0, 1, \ldots, n-1\} \to \mathsf{nat})$$

$$\land (0 \le i \le n \land (\forall j \bullet 0 \le j < i \Rightarrow a.j \neq x))$$

$$\land (i < n \land a.i = x)$$

Intuitively the execution of an invariant diagram starts from an initial situation and follows the transitions which are *enabled*. At each step the invariant of the current situation must be satisfied by the current value of the program variables. The execution terminates in a situation i when i is reached, and there are no enabled transitions from i.

The formal definition of an invariant diagram requires that there must be a transition between any two situations. However in our example (1) this requirement is not meet, there is no transition between situation 3 and 4. Formally, when there is no transition between two situations, we assume that there is a default transition ($\mathsf{miracle} = [\mathsf{false}]$) which is always disabled. Always, when we draw the diagram, we omit the transitions labeled by $\mathsf{miracle}$.

Invariant programs are more general than imperative programs, they can be thought of as multiple entry, multiple exist programs. In principle an invariant program could start and terminate in any situation. If the program represented in (1) starts in situation 1, then it can terminate in situations 2 if the element x is not member of the array a or in situation 3 otherwise.

3 Operational semantics

We introduce in this section smallstep and bigstep operational semantics for invariant diagrams and we prove their equivalence.

3.1 Small step semantics

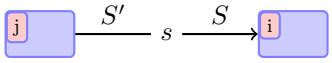
We introduce first the smallstep semantics for transitions. If $S, T \in \mathsf{Trs}$ and $s, s' \in \Sigma$, then the *smallstep relation* $(s, S) \rightarrow (s', T)$ is true if from state s we get to s' by executing one *step* (atomic statement) of S, and T is the transition S from which the executed step is removed. If the transition S consists of only one step, then the smallstep relation becomes $(s, S) \rightarrow (s', [])$. We denote by $(s, S) \rightarrow \bot$ the fact that the execution of S fails in the next step when starting from s.

$$\frac{b.s}{(s, \{b\}) \rightarrow (s, [])} \qquad \frac{\neg b.s}{(s, \{b\}) \rightarrow \bot} \qquad \frac{b.s}{(s, [b]) \rightarrow (s, [])}$$

$$\frac{R.s.s'}{(s, [R]) \rightarrow (s', [])} \qquad \frac{}{(s, S \sqcap T) \rightarrow (s, S)} \qquad \frac{}{(s, S \sqcap T) \rightarrow (s, T)}$$

$$\frac{(s, S) \rightarrow (s', S')}{(s, S\,;\, T) \rightarrow (s', S'\,;\, T)} \qquad \frac{(s, S) \rightarrow (s', [])}{(s, S\,;\, T) \rightarrow (s', T)} \qquad \frac{(s, S) \rightarrow \bot}{(s, S\,;\, T) \rightarrow \bot}$$

Let D be a transition diagram. Figure (2) represents one transition of D labeled by $S'\,;\, S$. We assume that the execution reached the state s in this transition. Then the tuple (s, S, i, D) denotes the status of the execution. The execution is in state s, and it proceeds towards the situation i by executing S. If the execution reaches i in a state s', then status of the execution is denoted by $(s', [], i, D)$.



(2)

The *smallstep relation* $(s, A, i, D) \to (s', B, i, D)$, where $A, B \in \mathsf{Trs} \cup \{[]\}$, is defined by the following rules.

$$\frac{(s, D_{i,j}) \to (s', S)}{(s, [], i, D) \to (s', S, j, D)} \qquad \frac{(s, S) \to (s', S')}{(s, S, i, D) \to (s', S', i, D)}$$

The transition diagram could *fail* in (s, A, i, D), denoted by $(s, S, i, D) \to \bot$, if some available transition could fail in next step.

$$\frac{(s, D_{ij}) \to \bot}{(s, [], i, D) \to \bot} \qquad \frac{(s, S) \to \bot}{(s, S, i, D) \to \bot}$$

3.2 Big step semantics

Similarly to smallstep semantics, we introduce first the bigstep semantics of transitions. If $S \in \mathsf{Trs}$ and $s, s' \in \Sigma$, then the *bigstep relation* $(s, S) \rightsquigarrow s'$ is true if there is an execution of S starting in s and ending in s'. $(s, S) \rightsquigarrow s'$ is defined by induction on the structure of S.

$$\frac{b.s}{(s, \{b\}) \rightsquigarrow s} \qquad \frac{b.s}{(s, [b]) \rightsquigarrow s} \qquad \frac{R.s.s'}{(s, [R]) \rightsquigarrow s'}$$

$$\frac{(s, S) \rightsquigarrow s'}{(s, S \sqcap T) \rightsquigarrow s'} \qquad \frac{(s, T) \rightsquigarrow s'}{(s, S \sqcap T) \rightsquigarrow s'} \qquad \frac{(s, S) \rightsquigarrow s' \wedge (s', T) \rightsquigarrow s''}{(s, S\,;\, T) \rightsquigarrow s''}$$

A transition S, starting from a state s, may *fail* (denoted $(s, S) \rightsquigarrow \bot$) if some of its executions leads to a false assertion.Failure is defined by induction on the structure of S.

$$\frac{\neg b.s}{(s, \{b\}) \rightsquigarrow \bot} \qquad \frac{(s, S) \rightsquigarrow \bot}{(s, S \sqcap T) \rightsquigarrow \bot} \qquad \frac{(s, T) \rightsquigarrow \bot}{(s, S \sqcap T) \rightsquigarrow \bot}$$

$$\frac{(s, S) \rightsquigarrow \bot}{(s, S\,;\, T) \rightsquigarrow \bot} \qquad \frac{(s, S) \rightsquigarrow s' \wedge (s', T) \rightsquigarrow \bot}{(s, S\,;\, T) \rightsquigarrow \bot}$$

Similarly, the execution of S, starting from a state s, is *miraculous* or *disabled* (denoted $(s, S) \rightsquigarrow \top$) if any of its executions leads to a false assumption or to a demonic update $[R]$ which *cannot progress*. The demonic update $[R]$ *cannot progress* from a state s if for all states s', $R.s.s'$ is false.

$$\frac{\neg b.s}{(s, [b]) \rightsquigarrow \top} \qquad \frac{\forall s' \bullet \neg R.s.s'}{(s, [R]) \rightsquigarrow \top} \qquad \frac{(s, S) \rightsquigarrow \top \wedge (s, T) \rightsquigarrow \top}{(s, S \sqcap T) \rightsquigarrow \top}$$

$$\frac{(s, S) \rightsquigarrow \top}{(s, S\,;\, T) \rightsquigarrow \top} \qquad \frac{(s, S) \not\rightsquigarrow \bot \wedge (\forall s' \bullet (s, S) \rightsquigarrow s' \Rightarrow (s', T) \rightsquigarrow \top}{(s, S\,;\, T) \rightsquigarrow \top}$$

Theorem 1 *Miracle could be defined in terms of bigstep and fail.*

$$(s, S) \rightsquigarrow \top \Leftrightarrow ((s, S) \not\rightsquigarrow \bot \wedge (\forall s' \bullet (s, S) \not\rightsquigarrow s'))$$

If $D \in I \times I \rightarrow \mathsf{Trs}$, $s, s' \in \Sigma$, and $i, j \in I$, then the *bigstep relation* $(s, i, D) \rightsquigarrow (s', j)$ is true if there is an execution from state s and situation i, following the enabled transitions D, ending in state s' and situation j, and all transitions from state s' and situation j are disabled. The execution of D from state s and situation i *may fail*, denoted $(s, i, D) \rightsquigarrow \bot$, if there is a situation j such that the transition $D_{i,j}$ may fail when starting from s.

$$\frac{(s, D_{i,j}) \rightsquigarrow s' \wedge (s', j, D) \rightsquigarrow (s'', k)}{(s, i, D) \rightsquigarrow (s'', k)}$$

$$\frac{(\forall j \bullet (s, D_{i,j}) \rightsquigarrow \top)}{(s, i, D) \rightsquigarrow (s, i)} \qquad \frac{(s, D_{i,j}) \rightsquigarrow \bot}{(s, i, D) \rightsquigarrow \bot}$$

When starting from state s and situation i, the transition diagram T *terminates*, denoted $(s, i, T) \downarrow$, if all execution paths starting in s, i are finite and do not fail.

$$\frac{(\forall j \bullet (s, D_{i,j}) \rightsquigarrow \top)}{(s, i, D) \downarrow}$$

$$\frac{(s, i, D) \not\rightsquigarrow \bot \wedge (\forall j, s' \bullet (s, D_{i,j}) \rightsquigarrow s' \Rightarrow (s', j, D) \downarrow)}{(s, i, D) \downarrow}$$

The bigstep semantics is useful in establishing further properties of transition diagrams, however it does not give a very intuitive understanding of how invariant diagrams are executed. In the next section we introduce the smallstep operational semantics for transition diagrams which is closer to the way the diagrams would be executed by a computer.

3.3 Connection between bigstep semantics and smallstep semantics.

The small step semantics is equivalent to the bis step semantics in the following sense. If the execution of D starts from a state s and a situation i and proceeds in small steps until a state s' in situation j, and there are no transitions enabled from (s', j), then this is equivalent with D performing a big step from (s, i) to (s', j). In the next theorems the symbol $\xrightarrow{*}$ denotes the reflexive and transitive closure of the relation $\rightarrow$.

Theorem 2 $(s, S) \rightsquigarrow s' \Leftrightarrow (s, S) \xrightarrow{*} (s', [])$

Theorem 3 $(s, S) \rightsquigarrow \bot \Leftrightarrow (s, S) \xrightarrow{*} \bot$

We define the miracle in the smallstep semantics by
(s, S) $\dashrightarrow \top = \neg(s, S) \xrightarrow{*} \bot \wedge (\forall s' \bullet \neg(s, S) \xrightarrow{*} (s', []))$

Theorem 4

$$(s, i, D) \rightsquigarrow (s', j) \Leftrightarrow (s, [], i, D) \xrightarrow{*} (s', [], j, D) \wedge (\forall k \bullet (s', D_{j,k}) \dashrightarrow \top)$$

In the remainder of this paper we will work with bigstep semantics only.

4 Weakest precondition and predicate transformers

Proving correctness of invariant diagrams is unfeasible using the operational semantics. We will therefore define here a compositional semantics for invariant based programs, based on the notion of weakest preconditions.

4.1 Weakest precondition and predicate transformers for transitions.

If $p, q \in \mathsf{Pred}$, and $S \in \mathsf{Trs}$ then the *Hoare total correctness triple* $p \{\![S]\!\} q$ denotes the fact that if the transition S start in state s from p, then it terminates in a state from q. The Hoare triple $p \{\![S]\!\} q$ is *valid*, denoted $\models p \{\![S]\!\} q$, if

$$\models p \{\![S]\!\} q \Leftrightarrow (\forall s \bullet p.s \Rightarrow (s, S) \not\rightsquigarrow \bot \wedge (\forall s' \bullet (s, S) \rightsquigarrow s' \Rightarrow q.s')) \qquad (3)$$

The *weakest precondition* for a transition S and a *post condition* q is a predicate, $\mathsf{wp}.S.q \in \mathsf{Pred}$. For a state s, $\mathsf{wp}.S.q.s$ is true if the execution of S does not fail and always terminates in a state s' from q ($q.s'$ is true). Using the bigstep operational semantics for transitions we define the weakest precondition by:

$$\mathsf{wp}.S.q.s = (s, S) \not\rightsquigarrow \bot \wedge (\forall s' \bullet (s, S) \rightsquigarrow s' \Rightarrow q.s').$$

The validity of Hoare triples could be expressed equivalently using the weakest precondition:

$$\models p \{\![S]\!\} q \Leftrightarrow p \subseteq \mathsf{wp}.S.q \qquad (4)$$

Relation (4) reduces the proof of validity of a Hoare triple to an inclusion of predicates. However the predicate $\mathsf{wp}.S.q$ is defined in terms of bigstep semantics, and the proof of the statement $p \subseteq \mathsf{wp}.S.q$ is still unfeasible in practice.

For $S \in \mathsf{Trs}$ we define, by induction on S, the *predicate transformer* associated to S, $\mathsf{pt}.S : \mathsf{Pred} \rightarrow \mathsf{Pred}$ by:

$$
\begin{array}{lcl}
\mathsf{pt}.\{p\}.q & = & p \wedge q \\
\mathsf{pt}.[p].q & = & \neg p \vee q \\
\mathsf{pt}.[R].q.s & = & (\forall s' \bullet R.s.s' \Rightarrow q.s') \\
\mathsf{pt}.(S \sqcap T).q & = & \mathsf{pt}.S.q \wedge \mathsf{pt}.T.q \\
\mathsf{pt}.(S\,;\,T).q & = & \mathsf{pt}.S.(\mathsf{pt}.T.q)
\end{array}
$$

Theorem 5 *For all* $S \in \mathsf{Trs}$

$$\mathsf{wp}.S = \mathsf{pt}.S$$

Proof. By induction on the structure of S. ■

Using Theorem 5 and relation (4) it follows

$$\models p \,\{\!|\, S \,|\!\}\, q \Leftrightarrow p \subseteq \mathsf{pt}.S.q \tag{5}$$

The relation (5) reduces the proof of the validity of a Hoare triple to an inclusion of predicates. These predicates are defined in terms of the predicates p, q, the predicates and expressions occurring in S using Boolean connectives ($\wedge, \vee, \rightarrow$, . . .).

Theorem 6 *For all* $S \in \mathsf{Trs}$ *the predicate transformer* $\mathsf{pt}.S$ *is monotonic.*

Proof. This fact follows directly from Theorem 5 and the definition of $\mathsf{wp}.S$. ■

The *guard* of a transition S is a predicate denoted $\mathsf{grd}.S \in \mathsf{Pred}$ and is true for all states s from which the execution of S is *enabled*.

$$\mathsf{grd}.S = \neg \mathsf{pt}.S.\mathsf{false}$$

Theorem 7 *The guard of a transition* S *is true in a state* s *if and only if the execution of* S *starting from* s *is not miraculous:*

$$\mathsf{grd}.S.s \Leftrightarrow (s, S) \not\rightarrowtail \top$$

Proof. Using Theorem 1, Theorem 5, and the definitions of grd and wp.

$\mathsf{grd}.S.s$

$=$ {Definition of grd}

$\neg \mathsf{pt}.S.\mathsf{false}$

$=$ {Theorem 5}

$\neg \mathsf{wp}.S.\mathsf{false}$

$=$ {Definition of wp}

$\neg((s, S) \not\rightsquigarrow \bot \wedge (\forall s' \bullet (s, S) \rightsquigarrow s' \Rightarrow \mathsf{false}.s'))$

$=$ {Boolean properties}

$\neg((s, S) \not\rightsquigarrow \bot \wedge (\forall s' \bullet (s, S) \not\rightsquigarrow s'))$

$=$ {Theorem 1}

$(s, S) \not\rightsquigarrow \top$ ■

4.2 Weakest precondition and predicate transformers for transition diagrams

The Hoare triples for diagrams have similar interpretations to those of the transitions. However, a diagram may be executed starting in any situation and it may terminate in any situation. Let $P, Q : I \to \mathsf{Pred}$ and $D : I \times I \to \mathsf{Pred}$. The *diagram Hoare total correctness triple*, $P \{\!|\, D \,|\!\} Q$, is true if whenever the execution of D starts in a state s from a situation i, such that $P.i.s$ is true, then D always terminates, and if D terminates in a state s' and a situation j, then $Q.j.s'$ is true. The predicate $P.i$ is the *precondition* of D when starting from situation i. Similarly, $Q.j$ is the *postcondition* of D when terminating in situation j.

The Hoare triple $P \{\!|\, D \,|\!\} Q$ is *valid*, denoted $\models P \{\!|\, D \,|\!\} Q$, if

$$\begin{array}{ll} & \models P \{\!|\, D \,|\!\} Q \\ \Leftrightarrow & \\ & (\forall i, s \bullet P.i.s \Rightarrow (s, i, D) \downarrow \wedge (\forall j, s' \bullet (s, i, D) \rightsquigarrow (s', j) \Rightarrow Q.j.s')) \end{array} \tag{6}$$

The *weakest precondition* for a diagram D and a postcondition Q is an indexed predicate $\mathsf{wp}.D.Q : I \to \mathsf{Pred}$. For a state s and a situation i, $\mathsf{wp}.D.Q.i.s$ is true if the execution of D from $s,\ i$ always terminates, and if it terminates in a state s' and a situation j then $Q.j.s'$ is true. Using the bigstep operational semantics for diagrams we define the weakest precondition by:

$$\mathsf{wp}.D.Q.i.s = (s, i, D) \downarrow \wedge (\forall j, s' \bullet (s, i, D) \rightsquigarrow (s', j) \Rightarrow Q.j.s').$$

The validity of diagram Hoare triples could be expressed equivalently using the weakest precondition:

$$\models P \{\!|\, D \,|\!\} Q \Leftrightarrow P \subseteq \mathsf{wp}.D.Q \tag{7}$$

Relation (7) reduces the proof of validity of a Hoare triple to an inclusion of indexed predicates. However, similarly to transitions' case, proving $P \subseteq \mathsf{wp}.D.Q$ is unfeasible in practice due to the bigstep semantics expressions occurring in wp.

The *guard* of a situation i in a diagram D is a predicate $\mathsf{grd}.D.i \in \mathsf{Pred}$ which is true in those states in which the execution from situation i is enabled:

$$\mathsf{grd}.D.i = \bigvee_{j \in I} \mathsf{grd}.D_{i,j}$$

Let $\mathsf{Dpt} = (I \to \mathsf{Pred}) \to (I \to \mathsf{Pred})$. For $D \in I \times I \to \mathsf{Trs}$ let $F.D : \mathsf{Dpt} \to \mathsf{Dpt}$ be the monotonic function given by

$$F.D.U.Q.i.s = (\forall j \bullet \mathsf{pt}.D_{i,j}.(U.Q.j).s) \wedge (\neg \mathsf{grd}.D.i.s \Rightarrow Q.i.s)$$

The *predicate transformer* associated to D, $\mathsf{pt}.D : \mathsf{Dpt}$, is the least fix point of F:

$$\mathsf{pt}.D = \mu\, F.D$$

Theorem 8 $\mathsf{wp}.D = \mathsf{pt}.D$

Proof. We prove that $\mathsf{wp}.D$ is fixpoint for $F.D$ and it is smaller than any other fixpoint. ■

Using Theorem 8 and relation (7) it follows

$$\models P \,\{\![\, D \,]\!\}\, Q \Leftrightarrow P \subseteq \mathsf{pt}.D.Q \tag{8}$$

The relation (8) reduces the proof of the validity of a Hoare triple to an inclusion of predicates. However, unlike for transitions, the predicate $\mathsf{pt}.D.Q$ is a least fixpoint expression, and proving $P \subseteq \mathsf{pt}.D.Q$ is unfeasible in practice.

Theorem 9 *For all* $D \in I \times I \to \mathsf{Trs}$ *the predicate transformer* $\mathsf{pt}.D$ *is monotonic.*

Proof. This fact follows directly from Theorem 8 and the definition of $\mathsf{wp}.D$. ■

5 Axiomatic semantics

The weakest precondition semantics does not allow us to prove correctness of programs in practice, because of the use of the least fixed point operator. We need to define Hoare like proof rules for invariant based programs to establish correctness in practice.

5.1 Hoare rules for transitions

The Hoare triple $p \{\!|\ S\ |\!\} q$ is *correct*, denoted $\vdash p \{\!|\ S\ |\!\} q$, if it can be proved using following *Hoare rules*.

$$\frac{\forall s \bullet p.s \Rightarrow r.s \wedge q.s}{\vdash p \{\!|\ \{r\}\ |\!\} q} \qquad \frac{\forall s \bullet p.s \wedge r.s \Rightarrow q.s}{\vdash p \{\!|\ [r]\ |\!\} q}$$

$$\frac{\forall s, s' \bullet p.s \wedge R.s.s' \Rightarrow q.s'}{\vdash p \{\!|\ [R]\ |\!\} q} \qquad \frac{\vdash p \{\!|\ S\ |\!\} q \quad \vdash p \{\!|\ T\ |\!\} q}{\vdash p \{\!|\ S \sqcap T\ |\!\} q}$$

$$\frac{\vdash p \{\!|\ S\ |\!\} r \quad \vdash r \{\!|\ T\ |\!\} q}{\vdash p \{\!|\ S ; T\ |\!\} q} \qquad \frac{\vdash p \{\!|\ S\ |\!\} q \quad p' \subseteq p \wedge q \subseteq q'}{\vdash p' \{\!|\ S\ |\!\} q'}$$

The validity is equivalent to proving correctness using the Hoare rules, and, in practice, the Hoare rules are used to prove the correctness of transitions.

Theorem 10 *(Correctness)*

$$\vdash p \{\!|\ S\ |\!\} q \ \Rightarrow\ \models p \{\!|\ S\ |\!\} q$$

Proof. By induction on the structure of S. ■

Theorem 11

$$\mathsf{wp}.S.q \{\!|\ S\ |\!\} q.$$

Proof. We prove $\mathsf{pt}.S.q \{\!|\ S\ |\!\} q$ by induction on the structure of S. ■

Theorem 12 *(Completeness)*

$$\models p \{\!|\ S\ |\!\} q \ \Rightarrow \vdash p \{\!|\ S\ |\!\} q.$$

Proof. By the definition of $\models p \{\!|\ S\ |\!\} q$ and $\mathsf{wp}.q$ it follows $p \subseteq \mathsf{wp}.q$ and by theorem 11 and Hoare consequence rule it follows $p \{\!|\ S\ |\!\} q$. ■

Before introducing the proof rules for diagrams we need some definitions and properties of complete lattices and fixpoints.

5.2 Complete lattices and fixpoints

This section introduces some results about fixpoints in complete lattices [8]. These results are the main tools in proving correctness and completeness of the proof rules for invariant diagrams.

A *partially ordered* (*poset*) set $\langle L, \leq \rangle$ is a *complete lattice* if every subset of L has *least upper bound* or equivalently *greatest lower bound*. For a subset A of

L, $\vee A \in L$ denotes the least upper bound (*join*) of A and $\wedge A \in L$ denotes the greatest lower bound (*meet*) of A. If L is a complete lattice, than the *least* (*bottom*) and the *greatest* (*top*) elements of L exist and they are denoted by $\bot,\ \top \in L$, respectively. If A is a nonempty set and L is a lattice, than the *pointwise extension* of the order on L to $A \to L$ is also a complete lattice. The operations meet, join, bottom, and top on $A \to L$ are also the pointwise extensions of the corresponding operations on L. If $\langle A, \leq \rangle$ is a partially ordered set, then the set of *monotonic* functions from A to L, denoted $A \xrightarrow{m} L$ is also a complete lattice. The order, meet, join, top, and bottom on $A \xrightarrow{m} L$ are the pointwise extensions of the corresponding operations on L. For a complete lattice L, $\mathsf{MF}.L$ is the complete lattice of monotonic functions from L to L. The Boolean algebra with two elements Bool, the predicates Pred, the indexed predicates $I \to \mathsf{Pred}$, and the monotonic predicate transformers are complete lattices.

We list briefly some properties of well founded and well ordered sets that are needed in this paper. For a comprehensive treatment of this subject see [12]. A partially ordered set $\langle W, < \rangle$ is *well founded* if every nonempty subset of W has a *minimal element*. The poset $\langle W, < \rangle$ is *well ordered* if it is well founded and *total*.

Theorem 13 *For any set A there is a well ordered set $\langle W, < \rangle$ such that no function $f : W \to A$ is injective. In other words, for any function $f : W \to A$ there exists $w_1, w_2 \in W$, $w_1 < w_2$, such that $f.w_1 = f.w_2$. For a set A we denote by W_A a well ordered set satisfying the property above.*

Proof. This theorem follows from Cantor's theorem stating that the power set (set of all subsets) of any set A has a strictly greater cardinality than that of A and the well-ordering theorem stating that every set can be well-ordered. ■

We use Theorem 13 to give a new proof for the classical Knaster-Tarski fixpoint theorem [17]. We give a construction of the least fixpoint of a monotonic function on a complete lattice L based on a well ordered set. Our construction is more general than the one in [6] which is based on ordinals, since we only need a well ordered set.

Theorem 14 *If $\langle L, \leq \rangle$ is a complete lattice and $F : L \to L$ is a monotonic function, then F has a least fixpoint denoted by $\mu\, F$.*

Proof. Assume that $\langle L, \leq \rangle$ is a complete lattice and that $F : L \to L$ is a monotonic function. Let W_L be a well ordered set given by Theorem 13. Let $w \in W_L$ and define $x_w, x \in L$ by

$$x_w = \bigvee_{v<w} F.x_v \ \text{ and } \ x = \bigvee_{w \in W} x_w$$

Then x is the least fixpoint of F. We prove first a number of properties about x_w.

(a) We first prove that x_v forms an increasing chain:

$$v \leq w \Rightarrow x_v \leq x_w \tag{9}$$

Assume $v \leq w$. Then

$x_v \leq x_w$

$\Leftrightarrow$ {Definition of x_v and x_w}

$\bigvee_{u<v} F.x_u \leq \bigvee_{s<w} F.x_s$

$\Leftarrow$ {Properties of $\bigvee$, the least upper bound operator }

$\forall u \bullet u < v \Rightarrow (\exists s \bullet s < w \wedge F.x_u \leq F.x_s)$

$\Leftarrow$ {Existential quantifier introduction}

$\forall u \bullet u < v \Rightarrow u < w \wedge F.x_u \leq F.x_u$

$\Leftrightarrow$ {Assumption}

$\forall u \bullet u < v \Rightarrow F.x_u \leq F.x_u$

$\Leftrightarrow$ {Symmetry of $\leq$}

true

(b) We then prove that

$$x_w \leq F.x_w \tag{10}$$

The proof is as follows:

$x_w \leq F.x_w$

$\Leftrightarrow$ {Definition of x_w}

$\bigvee_{v<w} F.x_v \leq F.x_w$

$\Leftrightarrow$ {Definition of $\bigvee$}

$\forall v \bullet v < w \Rightarrow F.x_v \leq F.x_w$

$\Leftrightarrow$ {Monotonicity of F and property (9)}

true

(c) By the property of W_L it now follows that there exists $w_1 < w_2$ such that $x_{w_1} = x_{w_2}$. We now show that

$$x_{w_1} = F.x_{w_1} \tag{11}$$

We already proved $x_{w_1} \leq F.x_{w_1}$. The converse inequality is proved as follows:

x_{w_1}

$=$ {Assumption}

x_{w_2}

$=$ {Definition of x_{w_2}}

$\bigvee_{v<w_2} F.x_v$

$\geq$ {Definition of $\bigvee$ and $w_1 < w_2$}

$F.x_{w_1}$

(d) We now prove that

$$y = F.y \Rightarrow (\forall w \bullet x_w \leq y) \tag{12}$$

We prove (12) by well founded induction on w. Assume $y = F.y$ and $(\forall v \bullet v < w \Rightarrow x_v \leq y)$.

$x_w \leq y$

$\Leftrightarrow$ {Definition of x_w}

$\bigvee_{v<w} F.x_v \leq y$

$\Leftrightarrow$ {Definition of $\bigvee$}

$(\forall v \bullet v < w \Rightarrow F.x_v \leq y)$

$\Leftrightarrow$ {Assumption $y = F.y$}

$(\forall v \bullet v < w \Rightarrow F.x_v \leq F.y)$

$\Leftarrow$ {Monotonicity of F}

$(\forall v \bullet v < w \Rightarrow x_v \leq y)$

$\Leftrightarrow$ {Assumption}

true

We have now shown that the element x_{w_1} is a fixpoint for F and that for all $w \in W_L$, x_w is smaller than any fixpoint of F. It follows that $x = x_{w_1}$ and x is the least fixpoint of F . ■

Let $\langle W, < \rangle$ be a well founded set and $x_w \in L$ a collection of elements indexed by $w \in W$. Then the elements $x_{<w}, x \in L$ are given by

$$x_{<w} = \bigvee_{v<w} x_v \quad \text{and} \quad x = \bigvee_{w \in W} x_w$$

Theorem 15 *If $\langle L, \leq \rangle$ is a complete lattice, $F : L \to L$ is monotonic, and $x_w \in L$ is a collection of elements indexed by $w \in W$, then*

$$(\forall w \bullet x_w \leq F.x_{<w}) \Rightarrow x \leq \mu\, F$$

Proof. We prove by well founded induction that $(\forall w \bullet x_w \leq \mu\, F)$. ■

If $x, y \in L$, then $\alpha.(x, y) \in \mathsf{MF}.L$ is given by

$$\alpha.(x, y).z = \begin{cases} x & \text{if } z \geq y \\ \bot & \text{otherwise} \end{cases}$$

It is easy to prove that $\alpha.(x, y)$ is monotonic.

Lemma 16 *If $x, y \in L$, $x_i \in L$ for all $i \in I$, and $f \in \mathsf{MF}.L$, then*

$$\alpha.(x, y) \leq f \Leftrightarrow x \leq f.y$$

$$\alpha.(\vee x_i, y) = \vee \alpha(x_i, y)$$

Proof.

$\alpha.(x, y) \leq f$

$\Rightarrow$ {Definition of $\leq$ for functions}

$\alpha.(x, y).y \leq f.y$

$\Leftrightarrow$ {Definition of α}

$x \leq f.y$

The revers implication follows from. Assume $x \leq f.y$.

$\alpha.(x, y) \leq f$

$\Leftrightarrow$ {Definition of $\leq$}

$\forall z \bullet \alpha.(x, y).z \leq f.z$

$\Leftarrow$ {Generalization}

$\alpha.(x, y).z \leq f.z$

$\Leftrightarrow$ {Case $z \geq y$}

- $\alpha.(x, y).z \leq f.z$
  $[z \geq y]$
  $\Leftrightarrow$ {Definition of α}
  $x \leq f.z$
  $\Leftarrow$ {Monotonicity of f}
  $x \leq f.y$
  $\Leftrightarrow$ {Assumption}
  true
- $\alpha.(x, y).z \leq f.z$
  $[z \not\geq y]$
  $\Leftrightarrow$ {Definition of α}
  $\bot \leq f.z$
  $\Leftrightarrow$ {Definition of $\bot$}
  true

. . . true

The second property is proved by:

$\alpha.(\vee x_i, y) = \vee \alpha(x_i, y)$

$\Leftrightarrow$ {Function equality and definition of $\vee$ for functions}

$\forall z \bullet \alpha.(\vee x_i, y).z = \vee \alpha(x_i, y).z$

$\Leftarrow$ {Generalization}

$\alpha.(\vee x_i, y).z = \vee \alpha(x_i, y).z$

$\Leftrightarrow$ {Case $z \geq y$}

- $\alpha.(\vee x_i, y).z = \vee \alpha(x_i, y).z$
  $[z \geq y]$
  $\Leftrightarrow$ {Definition of α}
  $\vee x_i \leq \vee x_i$

$\Leftrightarrow$ {Reflexivity of $\leq$}

true

- $\alpha.(\vee x_i, y).z = \vee\alpha(x_i, y).z$

  $[z \not\geq y]$

$\Leftrightarrow$ {Definition of α}

$\bot \leq \vee\bot$

$\Leftrightarrow$ {Definition of $\bot$}

true

... true

Theorem 17 *If $x_w, y \in L$, and $F : \mathsf{MF}.L \rightarrow \mathsf{MF}.L$ is a monotonic function, then*

$$(\forall w \in W,\ f \in \mathsf{MF}.L \bullet x_{<w} \leq f.y \Rightarrow x_w \leq F.f.y) \Rightarrow x \leq (\mu\, F).y$$

Proof.

$(\forall w \in W,\ f \in \mathsf{MF}.L \bullet x_{<w} \leq f.y \Rightarrow x_w \leq F.f.y)$

$\Leftrightarrow$ {Lemma 16}

$(\forall w \in W,\ f \in \mathsf{MF}.L \bullet \alpha.(x_{<w}, y) \leq f \Rightarrow \alpha.(x_w, y) \leq F.f)$

$\Leftrightarrow$ {Monotonic function properties}

$(\forall w \in W \bullet \alpha.(x_w, y) \leq F.(\alpha.(x_{<w}, y)))$

$\Leftrightarrow$ {Lemma 16}

$(\forall w \in W \bullet \alpha.(x_w, y) \leq F.(\bigvee_{v<w} \alpha.(x_v, y)))$

$\Rightarrow$ {Theorem 15}

$\bigvee_{w\in W} \alpha.(x_w, y) \leq \mu\, F$

$\Leftrightarrow$ {Lemma 16}

$\alpha.(x, y) \leq \mu\, F$

$\Leftrightarrow$ {Lemma 16}

$x \leq (\mu\, F).y$ ■

5.3 Hoare rules for transition diagrams.

Let $\langle W, < \rangle$ be a well founded set, and $X_w : I \to \mathsf{Pred}$ a collection of indexed predicates for all $w \in W$. Then the indexed predicates $X_{<w},\ X : I \to \mathsf{Pred}$, are defined by

$$X_{<w} = \bigvee_{v<w} X_v, \quad X = \bigvee_{w \in W} X_w$$

The Hoare triple $P \{\!|\, D \,|\!\} Q$ is correct, denoted $\vdash\ P \{\!|\, D \,|\!\} Q$, if it can be proved using the following *Hoare rules*:

$$\frac{P' \subseteq P \quad Q \subseteq Q' \quad \vdash P \{\!|\, D \,|\!\} Q}{\vdash P' \{\!|\, D \,|\!\} Q'}$$

$$\frac{\forall i, j, w \bullet \vdash X_w.i \{\!|\, D_{i,j} \,|\!\} X_{<w}.j}{\vdash X \{\!|\, D \,|\!\} (X \wedge \neg \mathsf{grd}.D)}$$

Theorem 18

$$\vdash P \{\!|\, D \,|\!\} Q \ \Rightarrow\ \models P \{\!|\, D \,|\!\} Q$$

Proof. For all $i, j \in I$, we assume

$$\models X_w.i \{\!|\, D_{i,j} \,|\!\} X_{<w}.j \ (\Leftrightarrow X_w.i \subseteq \mathsf{pt}.D_{i,j}.(X_{<w}.j))$$

and we prove $\models X \{\!|\, D \,|\!\} X \wedge \neg \mathsf{grd}.D$ which is equivalent to

$$X \subseteq \mathsf{pt}.D.(X \wedge \neg \mathsf{grd}.D) = (\mu\, F.D).(X \wedge \neg \mathsf{grd}.D)$$

Using Theorem 17 we have to prove

$$X_{<w} \subseteq U.(X \wedge \neg \mathsf{grd}.D) \Rightarrow X_w \subseteq F.D.U.(X \wedge \neg \mathsf{grd}.D)$$

for all $w \in W$ and $U \in (I \to \mathsf{Pred}) \to (I \to \mathsf{Pred})$. We assume $X_{<w} \subseteq U.(X \wedge \neg \mathsf{grd}.D)$ and for $i \in I$ and $s \in \Sigma$ we assume $X_w.i.s$, and we prove $F.D.U.(X \wedge \neg \mathsf{grd}.D).i.s$.

- $F.D.U.(X \wedge \neg \mathsf{grd}.D).i.s$

$\Leftrightarrow$ {Definition of F}

$(\forall j \bullet \mathsf{pt}.D_{i,j}.(U.(X \wedge \neg \mathsf{grd}.D).j).s) \wedge (\neg \mathsf{grd}.D.i.s \Rightarrow (X \wedge \neg \mathsf{grd}.D).i.s)$

$\Leftrightarrow$ {Assumptions}

$(\forall j \bullet \mathsf{pt}.D_{i,j}.(U.(X \wedge \neg \mathsf{grd}.D).j).s)$

$\Leftarrow$ {Assumptions and monotonicity of $\mathsf{pt}.D_{i,j}$}

$(\forall j \bullet \mathsf{pt}.D_{i,j}.(X_{<w}.j).s)$

$\Leftarrow$ {Assumptions}

$X_w.i.s$

$\Leftrightarrow$ {Assumptions}

true ■

Theorem 19

$$\vdash \mathsf{wp}.D.Q \,\{\!|\, D \,|\!\}\, Q$$

Proof. We need to prove that there exists $X_w : I \to \mathsf{Pred}$ such that

$$\mathsf{wp}.D.Q \subseteq X$$

$$(X \wedge (\neg \mathsf{grd}.D)) \subseteq Q$$

$$(\forall i, j \in I,\ w \in W \bullet \vdash X_w.i \,\{\!|\, D_{i,j} \,|\!\}\, X_{<w}.j)$$

Let $T_w = \bigvee_{v<w} F.D.T_v$ and $X_w = T_w.Q$. By Theorem 14, $\bigvee_{w\in W} T_w$ is the least fixpoint of $F.D$, therefore $\mathsf{wp}.D = \bigvee_{w\in W} T_w$.

$\mathsf{wp}.D.Q \subseteq X$

$\Leftrightarrow$ {Assumptions}

$(\bigvee T_w).Q \subseteq \bigvee(T_w.Q)$

$\Leftrightarrow$ {Definition of $\bigvee$}

true

For the second property we have:

$(X \wedge (\neg \mathsf{grd}.D)) \subseteq Q$

$\Leftrightarrow$ {Definitions of X and $\bigvee$}

$(\forall w \bullet X_w \wedge (\neg \mathsf{grd}.D) \subseteq Q)$

$\Leftrightarrow$ {Definitions of X_w and $\bigvee$}

$(\forall w, v \bullet v < w \Rightarrow F.D.T_v.Q \wedge (\neg \mathsf{grd}.D) \subseteq Q)$

$\Leftarrow$ {Definition of $\subseteq$}

$\forall v, i, s \bullet F.D.T_v.Q.i.s \wedge \neg \mathsf{grd}.D.i.s \Rightarrow Q.i.s$

$\Leftrightarrow$ {Definition of $F.D$}

$\ldots \wedge (\neg \mathsf{grd}.D.i.s \Rightarrow Q.i.s) \wedge \neg \mathsf{grd}.D.i.s \Rightarrow Q.i.s$

$\Leftrightarrow$ {Boolean properties}

true

For the last property let $i, j \in I$ and $w \in W$.

$\vdash X_w.i \,\{\![\, D_{i,j} \,]\!\}\, X_{<w}.j$

$\Leftarrow$ {Theorem 12}

$\models X_w.i \,\{\![\, D_{i,j} \,]\!\}\, X_{<w}.j$

$\Leftrightarrow$ {Relation (4) }

$X_w.i \subseteq \mathsf{wp}.D_{i,j}.(X_{<w}.j)$

$\Leftrightarrow$ {Theorem 5}

$X_w.i \subseteq \mathsf{pt}.D_{i,j}.(X_{<w}.j)$

$\Leftrightarrow$ {Definition of X_w}

$T_w.Q.i \subseteq \mathsf{pt}.D_{i,j}.(X_{<w}.j)$

$\Leftrightarrow$ {Definition of T_w}

$(\bigvee_{v<w} F.D.T_v).Q.i \subseteq \mathsf{pt}.D_{i,j}.(X_{<w}.j)$

$\Leftrightarrow$ {For arbitrary v, $v < w$ and s}

$F.D.T_v.Q.i.s \Rightarrow \mathsf{pt}.D_{i,j}.(X_{<w}.j).s$

$\Leftarrow$ {Definition of $F.D$}

$(\forall j \bullet \mathsf{pt}.D_{i,j}.(T_v.Q.j).s) \Rightarrow \mathsf{pt}.D_{i,j}.(X_{<w}.j).s$

$\Leftarrow$ {Generalization}

$\mathsf{pt}.D_{i,j}.(T_v.Q.j).s \Rightarrow \mathsf{pt}.D_{i,j}.(X_{<w}.j).s$

$\Leftrightarrow$ {Definition of $X_{<w}$}

$\mathsf{pt}.D_{i,j}.(T_v.Q.j).s \Rightarrow \mathsf{pt}.D_{i,j}.((\bigvee_{u<w} T_u.Q).j).s$

$\Leftrightarrow$ {Monotonicity of $\mathsf{pt}.D_{i,j}$}

$\mathsf{pt}.D_{i,j}.(T_v.Q.j).s \Rightarrow (\exists u \bullet \mathsf{pt}.D_{i,j}.(T_u.Q.j).s)$

$\Leftarrow$ {Existential quantifier introduction}

$\mathsf{pt}.D_{i,j}.(T_v.Q.j).s \Rightarrow \mathsf{pt}.D_{i,j}.(T_v.Q.j).s$

$\Leftrightarrow$ {Symmetry of implication}

true

Theorem 20

$$\models P \{\![D]\!\} Q \;\Rightarrow\; \vdash P \{\![D]\!\} Q$$

Proof. This is a consequence of Theorem 19.

6 Conclusions

We have introduced in this paper the semantics and proof rules for invariant based programs. We have started by defining big step and small step operational semantics for transition diagrams and we proved their equivalence (a big step of the program is equivalent to a sequence of small steps until the execution is terminated). Using the big step operational semantics we have defined the weakest precondition of a transition diagram and we have proved that it is compositional (it can be computed from the post-conditions and transitions, using a fixpoint operator). Although the weakest precondition is compositional, it cannot be used directly to prove correctness for transition diagrams, due to the use of the least fixpoint operator. We therefore introduced total correctness Hoare proof rules for transition diagrams, and we proved that they are correct and complete with respect to the operational semantics. Both the correctness and the completeness of the proof rules for transition diagrams are consequences of more general results about least fixpoints of monotonic functions on complete lattices.

In addition to meeting our original challenge, we have also contributed to other areas of programming language semantics. We have given a sound and complete proof system for multiple-entry multiple-exit program statements with unrestricted flow of control and unbounded non-determinism. This is pretty much as general as you can get, without going into higher levels of modularity (procedures, data modules, classes, processes etc.). This gives us a very general framework for establishing soundness and completeness of proof system for simple imperative programs. Most programming languages can be seen as special cases of invariant based programs, with restricted flow of control. By mapping the control structures of such programming languages onto invariant based programs, it is easy for us to study the soundness of proof systems for these more restricted languages, by reducing their soundness to the soundness of invariant based programs (which has been proved). Our result also opens up the way for checking the correctness of more complex structures. Multiple exits will, in particular, be useful for modeling exception handling [2] in programming languages. Multiple entries can again be

used to model data modules (procedures with multiple entry points are an old trick for modeling data modules).

We proved all results presented in this paper using the PVS interactive theorem prover. This gives a very solid foundation of our results.

We are currently working on extensions of these results to procedures (with parameters and local variables), and to data refinement. Another direction of research is the specialization of the proof rule for termination of the execution of a transition diagram into a collection of rules that can be easier applied in practice.

References

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

[2] R. J. Back. Exception handling with multi-exit statements. In H. J. Hoffmann, editor, *6th Fachtagung Programmiersprachen und Programmentwicklungen*, volume 25 of *Informatik Fachberichte*, pages 71–82, Darmstadt, 1980. Springer-Verlag.

[3] R. J. Back. Semantic correctness of invariant based programs. In *International Workshop on Program Construction*, Chateau de Bonas, France, 1980.

[4] R. J. Back. Invariant based programs and their correctness. In W. Biermann, G Guiho, and Y Kodratoff, editors, *Automatic Program Construction Techniques*, number 223-242. MacMillan Publishing Company, 1983.

[5] R. J. Back. Invariant based programming revisited. In S. Donatelli and P.S. Thiagarajan, editors, *Petri Nets 2006, 27th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency*, volume 4024 of *Lecture Notes in Computer Science*, pages 1 – 18. Springer Verlag, Jun 2006.

[6] R.J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.

[7] A. Church. A formulation of the simple theory of types. *J. Symbolic logic*, 5:56–68, 1940.

[8] B.A. Davey and H.A. Priestley. *Introduction to lattices and order*. Cambridge University Press, New York, second edition, 2002.

[9] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976. With a foreword by C. A. R. Hoare, Prentice-Hall Series in Automatic Computation.

[10] E. C. R. Hehner. do considered od: A contribution to the programming calculus. *Acta Informatica*, 11(4):287–304, 1979.

[11] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[12] P.T. Johnstone. *Notes on logic and set theory*. Cambridge University Press, New York, NY, USA, 1987.

[13] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.

[14] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[15] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Clavert. PVS language reference. Technical report, Computer Science Laboratory, SRI International, dec 2001.

[16] J. C. Reynolds. Programming with transition diagrams. In D. Gries, editor, *Programming Methodology*. Springer Verlag, Berlin, 1978.

[17] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[18] M. H. Van Emden. Programming with verification conditions. *IEEE Trans. Softw. Eng.*, 5(2):148–159, 1979.

Turku
Centre *for*
Computer
Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2111-8
ISSN 1239-1891