Predicate Transformers and Higher Order Logic

R. J. R. Back

Computer Science Department
California Institute of Technology

Caltech-CS-TR-92-24

Predicate Transformers and Higher Order Logic

R.J.R. Back* J. von Wright
Åbo Akademi

Abstract

Predicate transformers are formalized in higher order logic. This gives a basis for mechanized reasoning about total correctness and refinement of programs. The notions of program variables and logical variables are explicated in the formalization. We show how to describe common program constructs, such as assignment statements, sequential and conditional composition, iteration, recursion, blocks and procedures with parameters, are described as predicate transformers in this framework. We also describe some specification oriented constructs, such as assert statements, guards and nondeterministic assignments. The monotonicity of these constructs over the lattice of predicates is proved, as well as the monotonicity of the statement constructors with respect to the refinement ordering on predicate transformers.

1 Introduction

1. Statements of a programming language can be given a semantics by associating every statement with a predicate transformer, i.e., a function mapping predicates to predicates. The *weakest precondition* semantics associates a statement with a predicate transformer with the following property: each postcondition is mapped to the weakest precondition that guarantees that the execution of the statement will terminate in a final state that satisfies the postcondition. This semantic interpretation of statements is useful for reasoning about total correctness and refinement of programs and specifications [5, 1].

The proofs used in such reasoning are usually semi-formal, done in the tradition of classical mathematics. This proof method generally works well, but there are situations when a higher level of formality is desirable. For example, reasoning about blocks with local variables is often done without an exact definition of the status of the local variables.

In this paper we show how reasoning in the weakest precondition framework can be given a solid logical foundation by using higher order logic (simple type theory) as a basis. We describe a programming notation that covers basic programming constructs, as well as blocks with local variables, recursion and procedures with parameters. Statements are predicate transformers, defined as terms of higher order logic. This formalization captures the weakest precondition semantics of the corresponding traditional programming notations.

An important property of statements in the weakest precondition calculus is monotonicity. All reasonable statements of a programming notation should denote monotonic predicate transformers. Statement constructors should also be monotonic with respect to the refinement relation on statements [1]. We prove that all the statement constructors introduced here have both these monotonicity properties.

2. One of our main motivations for this work is the desire to mechanize reasoning about programs, using a theorem prover based on simple type theory. One such prover is HOL [6], and we have admittedly been inspired by the HOL logic when we developed this theory. Our aim is

*This paper was prepared while the first author was on a sabbatical at Caltech, Pasadena to overcome some of the problems encountered in formalizing the theory of imperative languages using theorem provers [7, 6, 4].

The formalization of predicate transformers and refinement calculus as described here has in fact been implemented in HOL as a mechanized theory. The monotonicity results stated here have also all been constructed and checked in HOL.

3. The paper is organized as follows. In the next section, we give a very brief overview of higher order logic. In Section 3 we describe the basic ideas underlying our formalization of predicate transformers. Section 4 shows how to define basic program statements within this framework. Section 5 introduces some additional constructs, that are found useful in practice, and which can be defined in terms of the basic constructs. Section 6 proves that all the constructs introduced have the required monotonicity properties. Section 7 ends with a few comments and remarks.

2 Higher order logic

1. The logic assumed is a polymorphic higher order logic. We assume that there is a collection of basic types. Every type σ is interpreted as a set (also denoted σ). Examples of basic types are `bool` (the booleans), `num` (the natural numbers) and `int` (the integers). We adopt the convention that constants and type names are written in typewriter font.

We use traditional symbols for logical connectives. The boolean truth values are denoted `F` (falsity) and `T` (truth). The scope of binders ($\forall$, $\exists$ and λ) extends as far to the right as possible.

From the basic types we can form new types by repeatedly applying *type constructors*: we will need only product types $\sigma \times \tau$ and function types $\sigma \rightarrow \tau$, defined as usual. For a given type σ, the *predicate type* $\overline{\sigma}$ is defined by

$$\overline{\sigma} \overset{\text{def}}{=} \sigma \rightarrow \mathtt{bool}.$$

This type is so common in our treatment that it is convenient to have it as an abbreviation.

2. The elements of $\overline{\sigma}$ can also be interpreted as sets, by identifying a set with its characteristic function. Thus p is identified with the set

$$\{s | p\, s\}$$

Then we can write e.g., $\emptyset$ for `false` and $p \cup q$ for $p \wedge q$. We also have that $v \in p$ is equivalent to $p\,v$. We will use the predicate and the set notation interchangeably, choosing whichever is more convenient for the moment.

We also generalize the set notation in the following way: for arbitrary $q : \alpha \rightarrow \beta$ and $p : \alpha \rightarrow \mathtt{bool}$ the notation

$$\{q\, s | \, s : p\, s\}$$

(the set of all $q\,s$ where s ranges over all values such that $p\,s$ holds) stands for the corresponding characteristic function

$$\lambda s'.\, \exists s.\, p\, s \wedge (s' = q\, s)$$

3. In the HOL system, rigorous proofs are carried out within the framework of a sequent calculus. In order to make proofs shorter, we use an informal calculational proof style in this paper. However, all proofs are easily transformed into formal proofs.

Since the logic is higher-order, we permit quantification and lambda abstraction over arbitrary types. Functions can have arguments of any type. New constants can be introduced by simple definitions. When defining a function f we often write

$$f\, x \overset{\text{def}}{=} E$$

rather than the equivalent $f \stackrel{\text{def}}{=} \lambda x.\, E$. Note that in a definition such as (1), all free variables of E must occur free on the left hand side also.

4. We permit type variables α, β and γ in types. A type variable can be instantiated to any type (even to a type containing type variables). This means that we can define *polymorphic constants.* An example of a polymorphic constant is infix equality, with type

$$=: \alpha \rightarrow \alpha \rightarrow \texttt{bool}$$

(the fact that a term t has type σ is indicated by writing $t : \sigma$).

As another example, the everywhere false predicate is defined by pointwise extension:

$$\texttt{false} \quad \stackrel{\text{def}}{=} \quad (\lambda v : \alpha.\, \mathtt{F})$$

Note how powerful the definition of a polymorphic constant is: we have one single definition of **false**, but it has many possible instantiations. For example, instantiating the type variable to **int** gives **false** : $\overline{\texttt{int}}$.

3 Basic domains

1. A predicate on program variables is not the same thing as a boolean formula over program variables. In programming logic, these two things are often identified (or confused). This is the case in, e.g., Hoare logic, where program variables are free variables in formulas. A Hoare triple is written as $P\{S\}Q$, where P and Q are predicates and S is a statement. The Hoare triple

$$x \geq y\, \{x := x + 1\}\, x > y$$

does not identify the program variables are: x and y can be the program variables, or maybe x is the only program variable. In the latter case, we would have to interpret y as a logical variable (unless it happens to be some constant in the underlying domain). There may also be some other program variables, such as z, which happen not to occur in the pre-and postconditions, nor in the statement.

The merit of this approach is that it simplifies the notation and calculations. However, the advantage is often offset by the conceptual confusion that results from interpreting program variables as free variables. A typical case in point is the interpretation of logical variables. Often one assumes that there are two classes of variables, program variables and logical variables, and that different rules apply these two classes.

Below we show that by making the program variables explicit, a simple and precise treatment of program variables and logical variables in programming logics becomes possible. Program variables will correspond to bound variables in this approach, whereas logical variables will be free variables. This explains their different treatment in programming logics.

State spaces

2. A *state space* is a product type

$$\sigma = \sigma_1 \times \ldots \times \sigma_m, \tag{1}$$

where $m \geq 0$. A *state* in this state space is an element $(x_1, \ldots, x_m)$ of type σ. For $m = 0$, we have $\sigma = \texttt{unit}$, the trivial type with only one element.

3. Program variables are used in imperative languages to denote state components. The association of program variables with state components is done by a *declaration*, which we define to be a tuple

$$v = (x_1 : \sigma_1, \ldots, x_m : \sigma_m),$$

$m \geq 0$, of distinct variables x_i with associated types σ_i. The declaration v *declares* the state space $\sigma_v = \sigma_1 \times \ldots \times \sigma_m$, and associates variables names $x_1, \ldots, x_m$ with the state components.

Because higher order logic assumes that each variable in a term is associated with a type, we get declarations for free. The type of a variable may be explicitly indicated in a formula, or, if a type inference mechanism like the one in HOL is used, it can be left implicit in most cases.

4. We permit that the state space $\sigma = \sigma_1 \times \ldots \times \sigma_m$ contains type variables, and thus defines a polymorphic type. Hence, the theory of predicate transformers we describe here is *generic*, it is not bound to a specific choice of the program state space, nor to a specific number of state components.

As a special case, the state space may contain a single component. This means that any type can be instantiated for the state space, i.e., the approach we have taken does not even restrict the interpretation of states to be tuples.

Predicates

5. Let σ be a declaration. A *state predicate* on σ is a function

$$p : \sigma \rightarrow \mathtt{bool},$$

or $p : \overline{\sigma}$, using the abbreviation introduced earlier.

Let $P : \mathtt{bool}$ be a boolean term. Assume that we want to interpret P as a predicate on the program variables $v = (x_1 : \sigma_1, \ldots, x_m : \sigma_m)$. Then

$$\lambda v. P$$

is this state predicate. *Program variables* are thus bound in a predicate, while the free variables in the predicate $\lambda v. P$ are the *logical variables.* A state predicate is thus determined by a boolean term $P : \mathtt{bool}$ and a declaration v of program variables. The boolean term alone is not sufficient to determine a state predicate.

For instance, if P is $x + y < z + 1$ and the program variables are $(x : \mathtt{int}, y : \mathtt{int})$, this determines the state predicate

$$p \quad = \quad \lambda(x : \mathtt{int}, y : \mathtt{int}). x + y < z + 1.$$

Here z is free, and is thus a logical variable. The predicate p states that the sum of the first and the second state component is less than $z + 1$. The variables x and y are only used as local names for the state components, whereas z is some logical variable whose value is assumed to be determined by the environment. Assuming $z = 3$, the predicate p is true in state $(0, 1)$ but false in state $(2, 2)$.

6. A predicate $P : \mathtt{bool}$ with implicit program variables v determines the state predicate $\lambda v. P$. In the other direction, any state predicate $p : \overline{\sigma}$ can be described as a predicate with given implicit program variables v: we have by η-reduction that

$$p = \lambda v. p v$$

whenever v is a declaration of the state space σ. Here $pv : \mathtt{bool}$ is the predicate with implicit program variables v.

Manipulating state predicates

7. In order to express axioms and proof rules in programming logics, we need to manipulate predicates in different ways, such as renaming variables in predicates and substituting terms for free variables. We show below how to do these manipulations on state predicates.

8. Consider first renaming. Assume that $P : \mathtt{bool}$ is a predicate with implicit program variables v. We want to rename some of the program variables in P, changing the program variables v to w. The result is the predicate $P[w/v]$ with implicit program variables w.

Making the program variables explicit gives the state predicate $p = \lambda v.\, P$. The meaning of the state predicate does not depend on the way program variables are named. By α-conversion, we are therefore free to rename the program variables in predicates:

$$p = \lambda v.\, P = \lambda w.\, P[w/v].$$

9. Another important operation on predicates is *substitution*. Given a predicate P on implicit program variables v, we want to substitute the terms t for the program variables v. The result is denoted $P[t/v]$ in Hoare logic. When the program variables are explicit, the effect of substitution is achieved with β-reduction. Let again p be the corresponding state predicate. The result of the substitution is the state predicate $\lambda v.\, p\, t$,

$$\lambda v.\, p\, t \;=\; \lambda v.(\lambda v.\, P)t \;=\; \lambda v.\, P[t/v]$$

(Application is assumed to bind stronger than lambda-abstraction.)

For example, the result of substituting $x + y$ for x in the predicate $\lambda(x, y).\, x > y$ is denoted by the term:

$$\lambda(x, y).\, (\lambda(x, y).\, x > y)(x + y, y) \;=\; \lambda(x, y).\, x + y > y.$$

We use β-reduction to compute the result. Note how the application to $(x+y, y)$ shows explicitly that no substitution is made for y. Simultaneous substitution is also easily expressed in this way.

10. When program variables are implicit, it is easy to mix predicate expressions over different state spaces. For example, we can write

$$x < y \wedge x = z + 1$$

even though originally $x < y$ had state space (x, y) and $x = z+1$ had state space (x, z). Implicitly, we have projected both to the state space (x, y, z).

With state predicates, such a projection must be made explicit. Assume that v and w are declarations with $v \subseteq w$ (i.e., every component in v is also a component in w). If p is a predicate over the state declared by v, then the corresponding predicate over w is

$$\lambda w.\, p\, v$$

Consider as an example the state predicate $\lambda(x, y).\, x < y$. Projecting this on the larger state space (x, y, z) gives us the state predicate

$$\lambda(x, y, z).\, (\lambda(x, y).\, x < y)(x, y) = \lambda(x, y, z).\, x < y.$$

In a similar way, it is sometimes possible to project a predicate expression into a smaller state space (e.g., if we have quantified over or substituted for some variable). If $v \subseteq w$ and if p is a predicate over the state declared by w, such that p is independent of the variables in $u = w - v$ (i.e., that $(\forall u.\, p\, w) = p\, w$), then the corresponding predicate over v is

$$\lambda v.\, p\, w$$

In practice, we know that p is independent of u if no variables in u occur free in the predicate expression corresponding to p.

In fact, the operation $\lambda w.\, p\, v$ is meaningful also in the general case. It may be used to change some program variables to free variables and vice versa. The program variables in $v - w$ become logical variables, while the logical variables in $w - v$ become program variables.

Predicate lattice

11. The usual logical connectives are only defined for truth values, i.e., for boolean terms. We define the logical connectives for state predicates by pointwise extension of the corresponding operations on truth values. Let v be a declaration of state σ. We define the following operations on predicates on σ:

$$\begin{array}{rcl}
\texttt{false} & \overset{\text{def}}{=} & \lambda v.\,\mathrm{F} \\
\texttt{true} & \overset{\text{def}}{=} & \lambda v.\,\mathrm{T} \\
\neg p & \overset{\text{def}}{=} & \lambda v.\,\neg p\,v \\
p \wedge q & \overset{\text{def}}{=} & \lambda v.\,p\,v \wedge q\,v \\
p \vee q & \overset{\text{def}}{=} & \lambda v.\,p\,v \vee q\,v \\
p \Rightarrow q & \overset{\text{def}}{=} & \lambda v.\,p\,v \Rightarrow q\,v
\end{array}$$

The type bool is a two-element complete boolean lattice, with $\Rightarrow$ as the lattice ordering. We extend this ordering to state predicates, by defining.

$$p \leq q \quad \overset{\text{def}}{=} \quad \forall v.\,p\,v \Rightarrow q\,v$$

Note the difference between $\leq$ and $\Rightarrow$: the term $p \leq q$ has type bool while the term $p \Rightarrow q$ is a predicate.

The pointwise extension implies that every state predicate type $\overline{\sigma} = \sigma \rightarrow \texttt{bool}$ is also a complete boolean lattice, with the ordering $\leq$ on state predicates. The operations $\wedge$, $\vee$ and $\neg$ defined above for state predicates are the lattice meet, join and inverse operators, respectively. true is the top element while false is the bottom element of the state predicate lattice.

12. We can extend the definition of conjunction and disjunction of predicates to arbitrary conjunctions (meets) and disjunctions (joins) of predicates. If $A : \overline{\sigma} \rightarrow \texttt{bool}$ is (the characteristic function for) a set of predicates (i.e.,, $A : \overline{\overline{\sigma}}$), then we define

$$\begin{array}{rcl}
\bigwedge A & \overset{\text{def}}{=} & \lambda v.\,\forall p.\,A\,p \Rightarrow p\,v \\
\bigvee A & \overset{\text{def}}{=} & \lambda v.\,\exists p.\,A\,p \wedge p\,v
\end{array}$$

If the set A is given as $A = \{p_i | i \in I\}$, then we write $\bigwedge_{i \in I} p_i$ for $\bigwedge A$ (and similarly for joins). Note that $\bigwedge \emptyset = \texttt{true}$ and $\bigvee \emptyset = \texttt{false}$.

In the special case that $I = \alpha$ for some type α, we have that $A : \alpha \rightarrow \overline{\sigma}$. In that case, we can write the meet and join as

$$(\forall a : \alpha.\,A) = (\lambda v.\,\forall a : \alpha.\,A\,a\,v) \quad \text{and} \quad (\exists a : \alpha.\,A) = (\lambda v.\,\exists a : \alpha.\,A\,a\,v).$$

We do not define any operation corresponding to quantifying over program variables in a predicate. However, it is always possible to make the quantification inside the lambda abstraction of a predicate. Thus, universal quantification over x of the predicate $\lambda v.\,P$ yields the predicate $\lambda v.\,\forall x.\,P$.

State transformations and state relations

13. A *state transformation* is a function $f : \sigma \rightarrow \tau$. An example of a state function is

$$f \quad = \quad \lambda(x : \texttt{int}, y : \texttt{int}).\,(x + y, x - z).$$

Thus f will transform any state (x, y) to the new state $(x+y, x-z)$. Here z is a logical variable.

A *state relation* (or *nondeterministic state transformation*) is a function $r : \sigma \longrightarrow \overline{\tau}$. State relations can also be expressed $r : \overline{\sigma \times \tau}$, but we prefer the first typing, which emphasizes the interpretation of a relation as a function from initial states to sets of final states.

Predicate transformers

14. Let σ and τ be two state spaces. A *predicate transformer* is a function

$$s : \overline{\sigma} \longrightarrow \overline{\tau}.$$

Such a function maps predicates on σ to predicates on τ.

15. Operations on predicates are pointwise extended to predicate transformers, in the same way as operations on `bool` were extended to predicates. We define for predicate transformers $s, t : \overline{\sigma} \longrightarrow \overline{\tau}$ the operations

$$\begin{array}{rcl}
\texttt{abort} & \overset{\text{def}}{=} & \lambda q.\,\texttt{false} \\
\texttt{magic} & \overset{\text{def}}{=} & \lambda q.\,\texttt{true} \\
\neg s & \overset{\text{def}}{=} & \lambda q.\,\neg(s\,q) \\
s \wedge t & \overset{\text{def}}{=} & \lambda q.\, s\,q \wedge t\,q \\
s \vee t & \overset{\text{def}}{=} & \lambda q.\, s\,q \vee t\,q \\
s \Rightarrow t & \overset{\text{def}}{=} & \lambda q.\, s\,q \Rightarrow t\,q
\end{array}$$

16. If H is (the characteristic function for) a set of predicate transformers (i.e., $H : (\overline{\sigma} \longrightarrow \overline{\tau}) \longrightarrow$ `bool`), then we define

$$\begin{array}{rcl}
\bigwedge H & \overset{\text{def}}{=} & \lambda p.\, \bigwedge\{s\,p \mid s : s \in H\} \\
\bigvee H & \overset{\text{def}}{=} & \lambda p.\, \bigvee\{s\,p \mid s : s \in H\}
\end{array}$$

Both $\bigwedge$ and $\bigvee$ have type $\overline{\overline{\sigma} \longrightarrow \overline{\tau}} \longrightarrow \overline{\sigma} \longrightarrow \overline{\tau}$.

Again, in the special case that H is given as $H : \alpha \longrightarrow (\overline{\tau} \longrightarrow \overline{\sigma})$, we can define

$$(\forall a : \alpha.\, H\,a) = (\lambda q.\, \forall a : \alpha.\, H\,a\,q) \quad \textit{and} \quad (\exists a : \alpha.\, H\,a) = (\lambda q.\, \exists a : \alpha.\, H\,a\,q).$$

17. With these definitions $\overline{\sigma} \longrightarrow \overline{\tau}$ is a complete boolean lattice, for all σ and τ. The partial order induced by the lattice structure is the same as the pointwise extension of the ordering on predicates: for predicate transformers s and s' in $\overline{\sigma} \longrightarrow \overline{\tau}$, we define

$$s \leq s' \quad \overset{\text{def}}{=} \quad (\forall p.\, s\,p \leq s'\,p)$$

This is the refinement ordering introduced in [1].

18. Since predicate transformers are functions, functional composition is also defined for these. For predicate transformers $s : \overline{\sigma} \longrightarrow \overline{\tau}$ and $t : \overline{\tau} \longrightarrow \overline{\rho}$, the functional composition $s;t$ has type $\overline{\sigma} \longrightarrow \overline{\rho}$.

4 Statements as predicate transformers

1. Rather than introducing a separate programming language for the statements we are interested in, we will work directly with predicate transformers. Thus, we will define an algebra of predicate transformers, with constants corresponding to certain basic statements and operations on predicate transformers corresponding to structured statements. Ordinary programming notations, such as `skip` or $x := 0$ can then be seen as just textual abbreviation for specific predicate transformers.

The reason that we build the theory in this way is two-fold. First, by not fixing a specific programming language, our results will be independent of any particular language. We will define a number of useful operations on predicate transformers that can be used in any combination. There is no need to put all the constructs into a single language. The approach is also open-ended; we may add new operations to the algebra later, at will.

The other reason for this approach is that we can work directly with predicate transformers in higher order logic. Instead of working with two different domains, a syntactic domain of terms and a semantic domain of predicates and predicate transformers, we work directly with the semantics in higher order logic. The syntax of our terms is the syntax of higher order logic, while the semantics of our terms is given by the standard semantics of higher order logic. Hence, there is no need for us to give a separate semantic definition. Moreover, we also do not need to introduce a separate proof theory for statements, the proof theory for higher order logic is sufficient to reason about all aspects of our statements. The new constructs we need are introduced by definitions, so the resulting theory is a conservative extension of the basic theory of higher order logic. This guarantees that our theory is consistent.

This does not, of course, preclude us from proving a number of useful theorems about program statements, which in other approaches would be treated as defining axioms. In our approach, they will be theorems, and proving these theorems corresponds to giving a soundness proof of the axioms in other logics.

Below we define constants that will correspond to a basic set of program statements constructs. These constructs are complete, in the sense that any monotonic predicate transformer can be built out of these constructs. We refer to predicate transformers built as terms with these constructs as *statements.*

Assignment statements

2. The assignment statement is the cornerstone of any imperative programming language. It is also one of the most cumbersome constructs to get right. We define an assignment statement to be an operation of type

$$\mathbf{assign} : (\sigma \longrightarrow \tau) \longrightarrow (\overline{\tau} \longrightarrow \overline{\sigma}),$$

where

$$\mathbf{assign}\, e \overset{\text{def}}{=} \lambda q.\lambda v.\, q(e\, v).$$

Thus, an assignment statement takes a state transformer $e : \sigma \longrightarrow \tau$ and converts it to a predicate transformer in $(\overline{\tau} \longrightarrow \overline{\sigma})$. Note that the assignment statement may change the state space also, it is not restricted to only change the value of some state components.

3. Let us make things more concrete by an example. Consider the expression

$$\mathrm{wp}(x := x + y, P)$$

in Dijkstra's original notation. Let P be the predicate $x + y < z + 1$ and let the state space be $(x : \mathbf{int}, y : \mathbf{int})$. In our notation, the assignment statement is $\mathbf{assign}(\lambda(x, y).\,(x + y, y))$ and the predicate P is $\lambda(x,y).\,x+y < z+1$. We calculate the value of the predicate transformer expression using a sequence of beta conversions:

$$\begin{array}{rcl}
\mathrm{wp}(x := x+y, P) & \sim & \mathbf{assign}(\lambda(x,y).\,(x+y,y))(\lambda(x,y).\,x+y<z+1) \\
& = & \lambda(x,y).(\lambda(x,y).\,x+y<z+1)(\lambda(x,y).\,(x+y,y))(x,y) \\
& = & \lambda(x,y).(\lambda(x,y).\,x+y<z+1)(x+y,y) \\
& = & \lambda(x,y).x+y+y<z+1 \\
& = & \lambda(x,y).(x+y<z+1)[x+y/x] \\
& \sim & P[e/x]
\end{array}$$

4. We define the identity statement **skip** as

$$\mathbf{skip} = \mathbf{assign}(\lambda v.v).$$

By calculation, we have that

$$\begin{array}{l}
\quad \mathbf{skip}\, q \\
= \{\text{Definition}\} \\
\quad \mathbf{assign}(\lambda v.v)\, q \\
= \{\text{Definition}\} \\
\quad \lambda v.\, q((\lambda v.\, v)v) \\
= \{\text{Beta reduction}\} \\
\quad \lambda v.\, q\, v \\
= \{\text{Eta reduction}\} \\
\quad q
\end{array}$$

This shows that $\mathbf{skip} = \lambda q.\, q$.

Sequential and conditional composition

5. Sequential and conditional composition are defined as operations on predicate transformers, of the type

$$\begin{array}{rcl}
\mathsf{seq} & : & (\overline{\tau} \to \overline{\sigma}) \times (\overline{\rho} \to \overline{\tau}) \to (\overline{\rho} \to \overline{\sigma}) \\
\mathsf{cond} & : & \overline{\sigma} \to (\overline{\tau} \to \overline{\sigma}) \times (\overline{\tau} \to \overline{\sigma}) \to (\overline{\tau} \to \overline{\sigma})
\end{array}$$

defined by

$$\begin{array}{rcl}
\mathbf{seq}\,(s_1, s_2)q & \overset{\text{def}}{=} & s_1(s_2\, q) \\
\mathbf{cond}\, b\,(s_1, s_2)q & \overset{\text{def}}{=} & (b \wedge s_1\, q) \vee (\neg b \wedge s_2\, q).
\end{array}$$

6. To make example programs easier to read, we permit the following alternative notation for sequential and conditional composition:

$$\begin{array}{rcl}
s_1; s_2 & = & \mathbf{seq}\,(s_1, s_2) \\
b \to s_1 | s_2 & = & \mathbf{cond}\, b\,(s_1, s_2)
\end{array}$$

We can also permit a more traditional syntax for assignments. For example, we could write $x, z := x+y, x$ for the assignment $assign(\lambda(x,y,z).\,(x+y,y,x))$ above. However, for this way of writing (multiple) assignments to be meaningful, the following three conditions must be satisfied:

(i) The initial and final state spaces must be equal,

(ii) the declaration of the state tuple must be obvious from the context, and

(iii) the variable(s) on the left hand side of the assignment symbol must be in the state tuple.

Demonic and angelic nondeterminism

7. We have already defined meet and join operators on predicate transformers. For $s, t : (\overline{\tau} \rightarrow \overline{\sigma})$, we interpret

$$s \wedge t : (\overline{\tau} \rightarrow \overline{\sigma})$$

as a *demonic choice* between the two alternatives s and t, while

$$s \vee t : (\overline{\tau} \rightarrow \overline{\sigma})$$

is interpreted as an *angelic choice* between the two alternatives.

More generally, if S is a set of predicate transformers, we interpret $\bigwedge S$ as demonic choice and $\bigvee S$ as angelic choice between the statements in S. Intuitively, execution of $\bigwedge S$ is guaranteed to terminate in a final state where a given predicate q holds if and only if all of the statements in S do this. Dually, execution of $\bigvee S$ is guaranteed to terminate in a final state where q holds if and only if at least one of the statements in S does this.

8. The bottom of the predicate transformer lattice is

$$\texttt{abort} \overset{\text{def}}{=} \bigvee \emptyset$$

By definition, we have that $\texttt{abort}\, q = \texttt{false}$ for all predicates q. As a statement, `abort` corresponds to the always *nonterminating* program (it does not establish any postcondition).

The top of the predicate transformer lattice is

$$\texttt{magic} \overset{\text{def}}{=} \bigwedge \emptyset$$

Again, by definition, we have that $\texttt{magic}\, q = \texttt{true}$ for all predicates q. As a statement, `magic` corresponds to the *miraculous* program (it will establish any postcondition whatsoever).

Completeness

9. It turns out that the assignment statement, the sequential and conditional composition and the arbitrary meet and join are sufficient to express any monotonic predicate transformer [3]. In other words, any monotonic predicate transformer can be described as a term where only these constructs are employed. The converse also hold, as we will show in Section 6, so the set of statements coincides with the set of monotonic predicate transformers.

This characterization is infinitary in nature, because both meets and joins may contain an infinite number of components, and hence do not qualify as ordinary (finite) programming language constructs. However, this completeness result gives a nice and tight characterization of monotonic predicate transformers.

5 Derived constructs

1. The completeness result states that only the statement constructs defined above are really needed to describe a monotonic predicate transformer. From a practical point of view, we are, however, interested in a larger set of program constructs. Some infinitary constructs can, e.g., be expressed also in a finite way, such as nondeterministic assignments, recursion and iteration. In other cases, we need a convenient way of describing predicate transformers involving more than one state space. Below we define a collection of such useful constructs. We extend the notion of statement to cover also predicate transformers built using the constructs defined below.

Conditional constructs

2. Having permitted the nonterminating statement **abort** and the miraculous statement **magic**, we can introduce conditional nontermination (*assertions*) and conditional miracles (*guards*). We add the constants **assert** and **guard** to our notation:

$$\begin{aligned}\mathtt{assert}\, b &\overset{\text{def}}{=} b \longrightarrow \mathtt{skip}|\mathtt{abort}\\ \mathbf{guard}\, b &\overset{\text{def}}{=} b \longrightarrow \mathtt{skip}|\mathtt{magic}\end{aligned}$$

Both these constants have type $\overline{\sigma} \longrightarrow (\overline{\sigma} \longrightarrow \overline{\sigma})$ (note that since **skip** has type $(\overline{\sigma} \longrightarrow \overline{\sigma})$, τ in the type of **cond** must here be σ). From the definitions it follows that

$$\begin{aligned}\mathtt{assert}\, b\, q &= b \wedge q\\ \mathbf{guard}\, b\, q &= b \Rightarrow q\end{aligned}$$

We permit the abbreviations $\{b\}$ and $[b]$ for **assert** b and **guard** b, respectively.

3. Dijkstra's nondeterministic conditional composition can now be introduced by defining a new constant **if**:

$$\mathtt{if}\,(b_1, b_2)\,(s_1, s_2) \overset{\text{def}}{=} \{b_1 \vee b_2\}; ([b_1]; s_1 \wedge [b_2]; s_2)$$

We may use the abbreviation $\mathtt{if}\, b_1 \longrightarrow s_1 \,[]\, b_2 \longrightarrow s_2\ \mathtt{fi}$ for this construct.

Nondeterministic assignments

4. We may also define statements that change the state space in a nondeterministic way, according to some specified state relation. These statement are useful for specifying computations[1, 2, 8].

Let $r : \sigma \longrightarrow \overline{\tau}$ be a state relation. Then r can be viewed as a function from initial states to sets of possible final states. We want to capture the notion of a statement which chooses nondeterministically between these possible final states. Since the choice may be demonic or angelic, we have two forms of the statement. The *demonic assignment statement* is defined by

$$\mathtt{demass}\, r\, q \overset{\text{def}}{=} \lambda v.\, \forall v'.\, r\, v\, v' \Rightarrow q\, v',$$

while the *angelic assignment statement* is defined by

$$\mathtt{angass}\, r\, q \overset{\text{def}}{=} \lambda v.\, \exists v'.\, r\, v\, v' \wedge q\, v'$$

Both constructs have polymorphic type $(\sigma \longrightarrow \overline{\tau}) \longrightarrow (\overline{\tau} \longrightarrow \overline{\sigma})$. (Both these statements can also be defined in terms of the basic constructs of the previous section, but defining them directly gives a somewhat better feeling for their meaning).

5. As an example, consider the state relation $r = \lambda(x,y)(x',y').\, x' > x$, which specifies that the final value of x should be greater than the initial value (we prime the variables in the tuple that corresponds to the final state). The following calculation applies $\mathbf{demass}\, r$ to the predicate $q = \lambda(x,y).\, x > 1$:

$$\begin{array}{ll}
& \mathbf{demass}\, r\, q \\
= & \{\text{Definitions}\} \\
& \lambda(x,y).\, \forall(x',y').\, (\lambda(x,y)(x',y').\, x' > x)(x,y)(x',y') \Rightarrow (\lambda(x,y).\, x > 1)(x',y') \\
= & \{\text{Beta reduction}\} \\
& \lambda(x,y).\, \forall(x',y').\, x' > x \Rightarrow x' > 1 \\
= & \{\text{Arithmetic}\} \\
& \lambda(x,y).\, x \geq 1.
\end{array}$$

Hence, to guarantee that $x > 1$ holds in the final state, $x \geq 1$ must hold in the initial state.

6. Similar calculations for the corresponding angelic assignment give

$$\begin{array}{rcl}
\mathbf{angass}\, r\, (\lambda(x,y).\, x' = x+1) & = & \mathbf{true} \\
\mathbf{angass}\, r\, (\lambda(x,y).\, x' = x+2) & = & \mathbf{true}
\end{array}$$

This shows a characteristic feature of angelic nondeterminism. Loosely speaking, it anticipates the postcondition that we want established. Operationally, we can interpret this as parallel execution on a (possible infinite) number of separate identical copies of the state space. The postcondition is established, if one of these executions is guaranteed to establish it.

7. A convenient abbreviation for demonic and angelic assignment statements is as follows. Let $v = u, w$. Then define

$$\begin{array}{rcl}
u := u'.Q & \stackrel{\text{def}}{=} & \mathbf{demass}(\lambda(u,w)\,(u',w').\, Q \wedge w' = w) \\
u :\approx u'.Q & \stackrel{\text{def}}{=} & \mathbf{angass}(\lambda(u,w)\,(u',w').\, Q \wedge w' = w)
\end{array}$$

Both statements assign some values u' to u such that condition Q becomes satisfied, leaving w unchanged. In the first case, the choice when there is more than one possible assignment is demonic, in the second case the choice is angelic. Also, in the first case the result is miraculous termination if there is no possible assignment, in the second case the result is abortion. As was the case with the assignment statement, this abbreviation is meaningful only if the naming of the state components is clear from the context.

Blocks and local variables

8. Let $s : (\overline{\sigma \times \tau} \longrightarrow \overline{\sigma \times \tau})$ be a predicate transformer. We want the notation $\mathbf{block}\, s$ to correspond to a statement which adds *local variables* to the state space, then executes s and finally removes the variables from the state space. For this, we define the constant **block** as follows:

$$\mathbf{block}\, s = \mathbf{enter};\, s; \mathbf{exit}$$

where **enter** and **exit** are defined as follows. First, let us define the state relation $\mathbf{any} : \tau \longrightarrow \overline{\sigma \times \tau}$ and the state transformer $\mathbf{drop} : \sigma \times \tau \longrightarrow \tau$, by

$$\begin{array}{rcl}
\mathbf{any}\, u\, (v', u') & = & u = u' \\
\mathbf{drop}\, (v, u) & = & u.
\end{array}$$

Then, we define **enter** $: \overline{\sigma \times \tau} \to \overline{\tau}$ and **exit** $: \overline{\tau} \to \overline{\sigma \times \tau}$ by

$$\begin{aligned} \texttt{enter} &= \texttt{demass}\, \textit{any} \\ \texttt{exit} &= \texttt{assign}\, \textit{drop}. \end{aligned}$$

Thus, the **enter** statement adds a new state component v without changing the old state component u, choosing the value for the new state component arbitrarily. The **exit** statement simply drops the new state component, without changing the old state component.

The typing of `block` is as follows:

$$\texttt{block} : (\overline{\sigma \times \tau} \to \overline{\sigma \times \tau}) \to (\overline{\tau} \to \overline{\tau})$$

Note that the local variables are visible only in the type of `block`. For simplicity, we may permit the outfix notation

$$|[\ \textbf{var}\ u; s\ ]|$$

for `block` s, where the local variables are given by the declaration u.The role of u is to determine how the state space of s is divided up into local and global variables.

9. Using the definitions of the constructs involved, we can compute the predicate transformer for the block statement. We then get that

$$\texttt{block}\, s\, q = \lambda v. \forall u.\, s(\lambda(u,v).\, q\, v)(u,v) \qquad (2)$$

A more common definition of the block is $\text{wp}(|[\ \textbf{var}\, x; S\ ]|, Q) = \forall x.\text{wp}(S,Q)$, with the side condition that x must not be free in Q. Our definition needs no side condition, due to the type discipline.

Recursion and iteration

10. We may also construct new statements using recursion. In the next section we will show that all the predicate transformers that can be built using operations defined here are in fact monotonic. Since the monotonic predicate transformers of any given type form a complete lattice, we can define an operator μ such that if c is a unary predicate transformer constructor, then $\mu\, c$ is the least fixpoint of c. The definition is as follows:

$$\mu c \overset{\text{def}}{=} \bigwedge\{s \,|\, c\, s \leq s\}$$

The type of μ is

$$\mu : ((\overline{\tau} \to \overline{\sigma}) \to (\overline{\tau} \to \overline{\sigma})) \to (\overline{\tau} \to \overline{\sigma})$$

We permit the use of μ as a binder, writing $\mu X.\, H$ for $\mu(\lambda X.\, H)$.

11. We define iteration as a special case of recursion in the usual way. We define a constant **do** by

$$\textbf{do}\, b\, s \overset{\text{def}}{=} \mu X.\, (b \to s; X | \texttt{skip})$$

Straighforward calculations show that

$$\textbf{do}\, b\, s\, q = \mu x.\, (\neg b \wedge q) \vee (b \wedge s\, x)$$

which corresponds to the usual fixpoint definition of the semantics of iteration. We permit the traditional notation $\textbf{do}\, b \to s\ \textbf{od}$ for $\textbf{do}\, b\, s$.

A notation for procedures

12. The **let** construct is commonly used as a notational device in the λ-calculus: $\texttt{let}\,X = s\ \texttt{in}\ t$ indicates that X is used inside t as a name for s. We introduce this construct as an abbreviation:

$$\texttt{let}\,X = s\ \ \texttt{in}\ t \qquad \text{stands for} \qquad \texttt{apply}((\lambda X.\,t), s) \tag{3}$$

where

$$\texttt{apply}(f,t) = f\,t.$$

We have the typing

$$\texttt{apply} : ((\overline{\tau} \to \overline{\sigma}) \to (\overline{\tau'} \to \overline{\sigma'})) \times (\overline{\tau} \to \overline{\sigma}) \to (\overline{\tau'} \to \overline{\sigma'})$$

when X has predicate transformer type.

Parameterless procedure declarations are directly expressible with the **let** notation:

$$\begin{aligned} &\textbf{procedure}\,X = s; && (4)\\ &t && (5) \end{aligned}$$

corresponds to $\texttt{let}\,X = s\ \ \texttt{in}\ t$ when s is a statement.

By β-reduction, we have that $\texttt{let}\,X = s\ \texttt{in}\ t$ is the same as $t[s/X]$, which corresponds to the usual interpretation of a procedure without parameters.

Note that we do not define **let** as a constant. This is because its syntax obscures the fact that its real arguments are the function $\lambda X.t$ and the statement s. However, the rule (3) shows how every **let**-construct is translated into a corresponding **apply**-term, which in turn can be simplified into a straightforward substitution.

Procedures with parameters

13. Consider next procedures with parameters. We want to permit the notation

$$\texttt{let}\,X = |[\ \texttt{val}\ \ x; \texttt{res}\ \ y; s\ ]|\ \ \texttt{in}\ t \tag{6}$$

The declarations x and y indicate how the state space of s is to be partitioned into value parameters, result parameters and global variables. Here s must have type

$$s : (\overline{\sigma \times \sigma' \times \tau} \to \overline{\sigma \times \sigma' \times \tau})$$

where $x : \sigma$ and $y : \sigma'$.

14. A *call* has the form

$$X\,e\,e'$$

where the terms e and e' have the following types:

$$\begin{aligned} &e : \tau \to \sigma\\ &e' : \sigma \times \sigma' \times \tau \to \sigma \times \sigma' \times \tau \end{aligned}$$

The idea is that e is a state function which shows what value the value parameter is to get before execution of the body s, while e' indicates what is to happen with the result in the result parameter after execution. The example below shows the details of this.

The notation (6) is defined to be equal to the term

$$\texttt{apply}\,(\lambda X.\,t)\,s'$$

where s' is the following term:

$$s' \quad = \quad \lambda e\, e'.\, \texttt{block}\,(\texttt{assign}\,(\lambda(x,y,v).\,(e\,v,y,v))\,;\,s\,;\,\texttt{assign}\,e')$$

This term s' is substituted for X in t.

15. The following example shows the intended us of this parameterized procedure construct. We consider a state space declared as $(a : \texttt{int}, b : \texttt{int})$ and a procedure which squares an integer:

$$\begin{array}{l}\textbf{let}\, X = |[\ \texttt{val}\ x; \texttt{res}\ y; \texttt{assign}\, \lambda(x,y,a,b).(x,x^2,a,b)\ ]| \\ \ \texttt{in}\ \ \texttt{assign}(\lambda(a,b).(2,b));\ X(\lambda(a,b).a+1)(\lambda(x,y,a,b).(x,y,a,y))\end{array} \tag{7}$$

In a more conventional notation this would look as follows:

$$\begin{array}{l}\textbf{procedure}\, X(\texttt{val}\ x; \texttt{res}\ y)\, \textbf{is}\, y := x^2; \\ \textbf{begin}\, a := 2\,;\, X(a+1,b)\, \textbf{end}\end{array}$$

The effect is that a is assigned the value 2 and b the value 9. Note that in (7), the first argument in the call of X is

$$\lambda(a,b).\,a+1$$

which indicates that the value parameter is to get the value $a+1$ (i.e., the first component of the state plus one). The second argument in the call of X is

$$\lambda(x,y,a,b).\,(x,y,a,y)$$

which indicates that the result y (i.e., the second component in the state) is to be placed in the variable b (i.e., the fourth component in the state, which is the second component of the global state).

16. We can also permit a procedure to have only value parameters or only result parameters, with appropriate adaptions. In case a procedure has neither value nor result parameters, then its definition coincides with the definition of parameterless procedures given previously.

6 Monotonicity

1. A *statement* is a term of type $(\overline{\tau} \rightarrow \overline{\sigma})$ that is built using only the specific predicate transformers and predicate transformer constructors introduced in the previous sections: assignment, demonic and angelic assignment, sequential composition, conditional composition, meet and join, blocks, application, and fixpoint construction. A statement may contain free variables, which may also be of predicate transformer type.

In this section we want to show two things:

(i) that every statement is a monotonic predicate transformer, and

(ii) subcomponent replacement in statements is monotonic.

The former expresses the requirement of monotonicity with respect to the ordering on predicates. The latter means that if $s(t)$ is a predicate transformer with predicate transformer t as a component, then $t \leq t' \Rightarrow s(t) \leq s(t')$. This is the basic property needed for program refinement: replacing a component by its refinement gives a refinement of the whole statement.

Preliminary definitions

2. In order to express and prove the monotonicity properties for predicate transformers, it turns out to be convenient to introduce a more flexible notation for subtypes in higher order logic. Subtypes are not directly supported in the logic, but we can introduce notational conventions that mimic the use of subtypes.

Given a type α, a *subtype* α' of α denotes some subset $\alpha' = \{x \in \alpha \mid R(x)\}$. Here $R(x)$ is the characteristic predicate that the elements of α must satisfy in order to belong to the subtype. For simplicity, we identify the subtype with its characteristic function, writing α' for both.

3. We would like to use subtypes in our formulas in the same way as we use the ordinary types. This can be done, if we adopt the following conventions for typing:

$$\begin{array}{lll} x : \alpha' & \text{is correct when} & x : \alpha \text{ and } \alpha'(x) \\ x : \alpha' \times \beta' & \text{is correct when} & x : \alpha \times \beta \text{ and } \alpha'(\mathtt{fst}\, x) \wedge \beta'(\mathtt{snd}\, x) \\ f : \alpha' \to \beta' & \text{is correct when} & f : \alpha \to \beta \text{ and } (\forall x : \alpha'. \beta'(f(x))). \end{array}$$

These conventions are easily explained by considering types as sets. We have, e.g., that

$$x \in \alpha' \quad = \quad x \in \{y \in \alpha \mid \alpha'(y)\} \quad = \quad x \in \alpha \wedge \alpha'(x).$$

For the functional type, the convention expresses that the function on subtypes must be well defined, in the sense that if the argument is in subtype α', then the value must be an element of subtype β'.

With these conventions we can quantify over subtypes. For example, we have

$$\begin{array}{lll} \forall x : \alpha'. H & \text{stands for} & \forall x : \alpha. (\alpha'(x) \Rightarrow H) \\ \exists f : \alpha' \to \beta'. H & \text{stands for} & \exists f : \alpha \to \beta. (\forall x : \alpha. \alpha'(x) \Rightarrow \beta'(f(x)) \wedge H \end{array}$$

The conventions permit us to use subtypes as if they were ordinary types in our formulas. The difference to ordinary types is that we have to prove explicitly the associated closedness assumptions, since subtypes are only a notational abbreviation.

4. Let α and β be two types with associated partial orderings $\leq_\alpha$ and $\leq_\beta$. A function $f : \alpha \to \beta$ is *monotonic*, if

$$x \leq_\alpha x' \quad \Rightarrow \quad f(x) \leq_\beta f(x'), \tag{8}$$

for each $x, x' : \alpha$. We denote by $\alpha \to_m \beta$ the subtype of all monotonic functions $f : \alpha \to \beta$.

5. The subtype of $(\overline{\tau} \to \overline{\sigma})$ containing all *monotonic predicate transformers* is thus $(\overline{\tau} \to_m \overline{\sigma})$.

6. A *monotonic predicate transformer constructor* (on monotonic predicate transformers) is a function of the form

$$c : (\overline{\tau_1} \to_m \overline{\sigma_1}) \times \ldots \times (\overline{\tau_n} \to_m \overline{\sigma_n}) \to_m (\overline{\tau} \to_m \overline{\sigma}) \tag{9}$$

To prove that a constructor c is of this type, we need to show the following two properties (by the definition of subtypes given in the beginning):

$$\forall s_1, \ldots, s_n. (\bigwedge_{i=1}^{n} s_i \text{ is monotonic }) \quad \Rightarrow \quad c(s_1, \ldots, s_n) \text{ is monotonic} \tag{10}$$

$$\forall s_1, \ldots, s_n, t_1, \ldots, t_n. (\bigwedge_{i=1}^{n} s_i \leq t_i \quad \Rightarrow \quad c(s_1, \ldots, s_n) \leq c(t_1, \ldots, t_n) \tag{11}$$

We extend the ordering on predicate transformers pointwise to an ordering on monotonic predicate transformer constructors, by

$$c \leq c' \quad \stackrel{\text{def}}{=} \quad \forall s_1 : (\overline{\tau_1} \to_m \overline{\sigma_1}), \ldots, s_n : (\overline{\tau_n} \to_m \overline{\sigma_n}). \; c(s_1, \ldots, s_n) \leq c'(s_1, \ldots, s_n)$$

7. The Knaster-Tarski fixpoint theorem tells us that every monotonic predicate transformer on a fixed predicate space has a least fixpoint. The following definition defines a polymorphic fixpoint operator μ:

$$\mu f \quad \overset{\text{def}}{=} \quad \bigwedge\{p \mid f\,p \leq p\}$$

The type of μ is

$$\mu : (\alpha \to \alpha) \to \alpha$$

From the definition it is straighforward to prove the following two properties provided that f is monotonic:

$$\begin{array}{rcl} f(\mu f) & = & \mu f \\ \forall x.\, f\,x \leq x & \Rightarrow & \mu f \leq x \end{array}$$

This shows that μ is in fact the required least fixpoint operator.

Monotonicity of statements

8. Our main result is the following.

THEOREM 1 *Let $s : (\overline{\tau} \to \overline{\sigma})$ be a statement, and let $X_1 : (\overline{\tau_1} \to \overline{\sigma_1}), \ldots, X_n : (\overline{\tau_n} \to \overline{\sigma_n})$ be the free predicate transformer variables in s, $n \geq 0$. Then*

$$\lambda X_1 \ldots X_n . s : (\overline{\tau_1} \to_m \overline{\sigma_1}) \to_m \ldots (\overline{\tau_n} \to_m \overline{\sigma_n}) \to_m (\overline{\tau} \to_m \overline{\sigma}).$$

This expresses formally both requirement (i) and (ii) above, (i) being the special case where there are no free predicate transformer variables.

Proof The theorem is established by the lemmas proved below, as any statement can be built from assignment and the two kinds of nondeterministic assignment using sequential and conditional composition, nondeterministic choice, the block construction, recursion and functional application. Q.E.D

9. The following lemma establishes the monotonicity of assignment statements.

LEMMA 1 **assign** $e : (\overline{\sigma} \to_m \overline{\tau})$, *for any* $e : \sigma \to \tau$.

Proof

$$\begin{array}{ll} & p \leq q \\ = & \{\text{Definition of predicate ordering}\} \\ & \forall v.\, p\,v \Rightarrow q\,v \\ \Rightarrow & \{\text{Property of universal quantification}\} \\ & \forall v.\, p(e\,v) \Rightarrow q(e\,v) \\ = & \{\text{Definition of ordering of predicates}\} \\ & (\lambda v.\, p(e\,v)) \leq (\lambda v.\, q(e\,v)) \\ = & \{\text{Definition of } \mathbf{assign}\} \\ & \mathbf{assign}\, e\, p \leq \mathbf{assign}\, e\, q \end{array}$$

Q.E.D

Note that we do not have to prove monotonicity of the `skip` statement, since it is defined as an `assign`-statement.

10. The monotonicity of sequential and conditional composition is established by the following lemma.

LEMMA 2 (i) $\mathsf{seq} : (\overline{\tau} \rightarrow_m \overline{\sigma}) \times (\overline{\rho} \rightarrow_m \overline{\tau}) \rightarrow (\overline{\rho} \rightarrow_m \overline{\sigma})$.

(ii) $\mathsf{cond}\, b : (\overline{\tau} \rightarrow_m \overline{\sigma}) \times (\overline{\tau} \rightarrow_m \overline{\sigma}) \rightarrow (\overline{\tau} \rightarrow_m \overline{\sigma})$, *for any* $b : \overline{\sigma}$.

Proof We only show the proof of (i), as the proof of (ii) is similar.

$$\begin{array}{ll} & p \leq q \\ \Rightarrow & \{\text{Assumption that } t \text{ is monotonic}\} \\ & t\,p \leq t\,q \\ \Rightarrow & \{\text{Assumption that } s \text{ is monotonic}\} \\ & s(t\,p) \leq s(t\,q) \\ = & \{\text{Definition of sequential composition}\} \\ & \mathsf{seq}\,(s,t)\,p \leq \mathsf{seq}\,(s,t)\,q \end{array}$$

Q.E.D

11. The monotonicity of sequential and conditional composition, viewed as predicate transformer constructors is established by the following lemma.

LEMMA 3 (i) $\mathsf{seq} : (\overline{\tau} \rightarrow_m \overline{\sigma}) \times (\overline{\rho} \rightarrow_m \overline{\tau}) \rightarrow_m (\overline{\rho} \rightarrow_m \overline{\sigma})$.

(ii) $\mathsf{cond}\, b : (\overline{\tau} \rightarrow_m \overline{\sigma}) \times (\overline{\tau} \rightarrow_m \overline{\sigma}) \rightarrow_m (\overline{\tau} \rightarrow_m \overline{\sigma})$, *for any* $b : \overline{\sigma}$.

Proof Assume that $s \leq s'$ and $t \leq t'$ and let q be an arbitrary predicate over σ.

Consider first sequential composition. Assume that s, s', t and t are monotonic, with $s \leq s'$ and $t \leq t'$. Starting from the trivial fact that $q \leq q$, we have

$$\begin{array}{ll} & q \leq q \\ \Rightarrow & \{\text{assumptions } t \leq t' \text{ and } s \text{ monotonic}\} \\ & s(t\,q) \leq s(t'\,q) \\ \Rightarrow & \{\text{transitivity of } \leq, \text{ since } s(t'\,q) \leq s'(t'\,q) \text{ by assumption}\} \\ & s(t\,q) \leq s'(t'\,q) \end{array}$$

As q was arbitrarily chosen, this shows that $s;t \leq s';t'$, as required.

For conditional composition, the proof is trivial. Q.E.D

12. From the theory of lattices it is well known that the monotonic functions form a complete sublattice of any function lattice. This means that the typings

$$\begin{array}{l} \bigwedge : ((\overline{\tau} \rightarrow_m \overline{\sigma}) \rightarrow \mathtt{bool}) \rightarrow (\overline{\tau} \rightarrow_m \overline{\sigma}) \\ \bigvee : ((\overline{\tau} \rightarrow_m \overline{\sigma}) \rightarrow \mathtt{bool}) \rightarrow (\overline{\tau} \rightarrow_m \overline{\sigma}) \end{array}$$

are correct. The subcomponent monotonicity property is proved in the following lemma:

LEMMA 4 *Let $S = \{s_i | i \in I\}$ and $T = \{t_i | i \in I\}$ be two sets of predicate transformers in $(\overline{\tau} \rightarrow_m \overline{\sigma})$. If $s_i \leq t_i$ holds for all $i \in I$, then*

$$\bigwedge_{i \in I} s_i \leq \bigwedge_{i \in I} t_i$$
$$\bigvee_{i \in I} s_i \leq \bigvee_{i \in I} t_i$$

Proof Both results follow directly from basic properties of meets and joins in complete lattices. Q.E.D

13. As a matter of fact, $\bigvee$ is also monotonic (and $\bigwedge$ antimonotonic) in the sense that it has type

$$\bigvee : ((\overline{\tau} \rightarrow_m \overline{\sigma}) \rightarrow \texttt{bool}) \rightarrow_m (\overline{\tau} \rightarrow_m \overline{\sigma})$$

since lifting the order on `bool` to sets yields the subset ordering. However, this monotonicity property is quite different from the subcomponent monotonicity property proved above.

Monotonicity of derived constructs

14. The monotonicity of the nondeterministic assignments are established by the following lemma.

LEMMA 5 *(i)* $\texttt{demass}\, r : (\overline{\tau} \rightarrow_m \overline{\sigma})$, *for any* $r : \sigma \rightarrow \overline{\tau}$, *and*

(ii) $\texttt{angass}\, r : (\overline{\tau} \rightarrow_m \overline{\sigma})$, *for any* $r : \sigma \rightarrow \overline{\tau}$.

Proof Assume that $p \leq q$ holds. Then

$$(\lambda v. \forall v'. r\, v\, v' \Rightarrow p\, v') \quad \Rightarrow \quad (\lambda v. \forall v'. r\, v\, v' \Rightarrow q\, v')$$

by the monotonicity of the logical operators. By the definition of `demass`, this proves the first case. The proof for `angass` is similar. Q.E.D

15. The following lemma shows that the block construct preserves monotonicity and that `block` is a monotonic constructor.

LEMMA 6 $\texttt{block} : (\overline{\sigma \times \tau} \rightarrow_m \overline{\sigma \times \tau}) \rightarrow_m (\overline{\tau} \rightarrow_m \overline{\tau})$

This result follows directly from the monotonicity of the constructs in terms of which the block construct is defined.

16. For the fixpoint operator, we can prove the following general result:

LEMMA 7 $\mu : ((\overline{\tau} \rightarrow_m \overline{\sigma}) \rightarrow_m (\overline{\tau} \rightarrow_m \overline{\sigma})) \rightarrow_m (\overline{\tau} \rightarrow_m \overline{\sigma})$

Proof First we have to show that if c is a function of type $(\overline{\tau} \rightarrow_m \overline{\sigma}) \rightarrow_m (\overline{\tau} \rightarrow_m \overline{\sigma})$, then $\mu\, c$ is in fact monotonic. This is a well-known fact from the theory of function lattices.

Now let c and d both have type $(\overline{\tau} \rightarrow_m \overline{\sigma}) \rightarrow_m (\overline{\tau} \rightarrow_m \overline{\sigma})$. Then

$$\begin{array}{ll} & c \leq d \\ \Rightarrow & \{\text{Definition of partial order}\} \\ & c(\mu d) \leq d(\mu d) \\ \Rightarrow & \{\mu d \text{ is fixpoint of } d\} \\ & c(\mu d) \leq \mu d \\ \Rightarrow & \{\mu c \text{ is least prefixed point of } c\} \\ & \mu c \leq \mu d \end{array}$$

which shows that μ is monotonic. Q.E.D

17. Assume that σ and τ are types with associated partial orders (both written $\leq$). Then application is monotonic for these types.

LEMMA 8 $\texttt{apply} : (\sigma \rightarrow_m \tau) \times \sigma \rightarrow_m \tau$

Proof Let $t, t' : \sigma$ and $f, f' : (\sigma \rightarrow_m \tau)$. Assume that $t \leq t'$ and $f \leq f'$. Then

$$\begin{array}{l} \quad f \leq f' \\ \Rightarrow \{\text{Definition of partial order, instantiation}\} \\ \quad f\,t \leq f'\,t \\ \Rightarrow \{\text{Assumption } t \leq t', \text{ monotonicity of } f', \text{ transitivity of } \leq\} \\ \quad f\,t \leq f'\,t' \end{array}$$

Q.E.D

18. The `let`-construct is monotonic, in the sense that the `apply`-construct that it stands for is monotonic. It immediately follows that procedure calls, which are described by `let`-constructs, are also monotonic.

7 Conclusions

1. We have shown how a programming and specification language based on a weakest precondition-semantics can be formulated in higher order logic. In particular, we showed how statements can be defined as terms in the logic. This means that we can use higher order logic as a logic for reasoning about imperative programs. Our syntax is slightly more complicated than ordinary syntax for simple imperative languages (in particular, this is the case for assignments and procedures with parameters). However, we have shown how more ordinary syntax can be permitted, using suitable abbreviations and conventions.

2. Our work can be used as a basis for mechanical reasoning about programs. Theorem provers based on simple type theory (such as the HOL proof assistant) can be used to formalize the programming notation we have developed. Reasoning about program correctness and refinement can then be carried out within higher order logic without the need for a separate proof theory.

3. Our notation permits predicate statements that map predicates over one state space to predicates over another space. In this paper, we have used this possibility to define special commands to entering and exiting a block with local variables. It can also be used for incorporating data refinement (data reification) into the notation, along the lines described in [3, 9]. This is done by introducing state transformations that transform a state of one (abstract) state space into a state of another (concrete) state space.

References

[1] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

[2] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[3] R. J. R. Back and J. von Wright. Refinement calculus I: Sequential nondeterministic programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, Lecture Notes in Computer Science, pages 42–66. Springer–Verlag, 1990.

[4] R. J. R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 1991.

[5] E. W. Dijkstra. *A Discipline of Programming.* Prentice–Hall International, 1976.

[6] Michael J. Gordon. Hol: A proof generating system for higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[7] I. A. Mason. Hoare's logic in the LF. Technical report 87-32, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1987.

[8] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[9] J. von Wright, J. Hekanaho, T. Langbacka, , and P. Luostarinen. Mechanizing some advanced refinement concepts. In Luc Claesen and Michael Gordon, editors, *Proceedings of the 1992 International Workshop on Higher Order Logic, Theorem Proving and its Applications*, pages 77 – 96. North-Holland, 1992.