He Jifeng, Oxford University Computing Laboratory, UK, John Cooke, Loughborough University, UK, and Peter Wallis, University of Bath, UK (Eds)

BCS-FACS 7th Refinement Workshop

Proceedings of the BCS-FACS 7th Refinement Workshop, Bath, 3-5 July 1996

Interpreting Nondeterminism in the Refinement Calculus (Invited Lecture)

R.J.R. Back and J. von Wright

ISBN 3-540-76104-7

Interpreting Nondeterminism in the Refinement Calculus

Ralph-Johan Back
Department of Computer Science, Abo Akademi University
Turku, Finland

Joakim von Wright
Department of Computer Science, Abo Akademi University
Turku, Finland

Abstract

We introduce a simple programming language and define its predicate transformer semantics. We motivate the choice of the constructs for the language by algebraic arguments, showing that the constructs are essentially the basic algebraic operations that are available for predicate transformers, when these are viewed as forming a complete lattice enriched category. We then show that the statements in the language can be given a simple operational interpretation, as describing the interaction between a user and a computing system. This gives a general intuitive interpretation of angelic and demonic nondeterminism. We also consider the notion of correctness and refinement of program statements that this intuitive interpretation gives rise to, and show the connection between the user-system interpretation and the interpretation of program execution as a game.

1 Introduction

We will define a simple programming language, intended to capture the interaction between a computing system and its user. We first describe the language formally (syntax and semantics) and motivate the choice of the language constructs by appealing to their fundamental algebraic properties. Our main interest here is, however, to give a simple intuitive interpretation of the programming language, and to motivate the notions of correctness and refinement for this language.

Consider the following simple language of *(program) statements*:

$$\begin{array}{rcll} S & ::= & \mathsf{abort} \mid \mathsf{skip} \mid \mathsf{magic} \mid \{p\} \mid [p] \mid \langle f \rangle \mid \{Q\} \mid [Q] \mid & (\textit{Basic statements}) \\ & & S_1 ; S_2 \mid S_1 \sqcup S_2 \mid S_1 \sqcap S_2 \mid (\sqcup i \in I \,.\, S_i) \mid (\sqcap i \in I \,.\, S_i) & (\textit{Compound statements}) \end{array}$$

A program statement is always associated with an initial state space Σ and a final state space Γ. These state spaces are types (we will later be more specific about what the state spaces are). The intuition is that execution of S always starts in some state $\sigma : \Sigma$ and if it terminates normally, then it terminates in a final state $\gamma : \Gamma$.

In the syntax above, $p : \Sigma \to \mathsf{Bool}$ is a *state predicate* that assigns a truth value to each state, $f : \Sigma \to \Gamma$ is a *state transformer* (a total function) that maps every state in Σ to a new state in Γ, while $Q : \Sigma \to \Gamma \to \mathsf{Bool}$ is a *state relation* that relates a state in Σ to zero, one or more new states in Γ.

The statements abort, skip and magic are special constants of the language. The compound constructs express a statement S in terms of component statements S_1, S_2 or in terms of a set $\{S_i \mid i \in I\}$ of statements.

1.1 Predicate transformer interpretation

Following Dijkstra [5], the statements are interpreted as *predicate transformers*, i.e., as functions that map a *postcondition* (a predicate on the final state space) to a *precondition* (a predicate on the initial state space). A predicate can be seen either as a function from states to truth values, or we can interpret it as a set of states (i.e., the set of states that are mapped to truth by the predicate). Below, we use the latter interpretation.

Let q be a predicate over the final state space Γ, and let σ be a state in the initial state space Σ. Furthermore, let true stand for the identically true predicate, i.e., the universal set Σ, and let false stand for the identically false predicate, i.e., the empty set $\emptyset$. We have the following definitions for the basic statements:

$$\begin{aligned}
\mathsf{abort}\, q &\overset{\mathrm{def}}{=} \mathsf{false} \\
\mathsf{skip}\, q &\overset{\mathrm{def}}{=} q \\
\mathsf{magic}\, q &\overset{\mathrm{def}}{=} \mathsf{true} \\
\{p\}\, q &\overset{\mathrm{def}}{=} p \cap q \\
[p]\, q &\overset{\mathrm{def}}{=} \neg p \cup q \\
\langle f \rangle\, q &\overset{\mathrm{def}}{=} f^{-1}\, q \\
\{Q\}\, q &\overset{\mathrm{def}}{=} \{\sigma \mid Q\,\sigma \subseteq q\} \\
[Q]\, q &\overset{\mathrm{def}}{=} \{\sigma \mid Q\,\sigma \cap q \neq \emptyset\}\ .
\end{aligned}$$

For the compound statements, we define

$$\begin{aligned}
(S_1; S_2)\, q &\overset{\mathrm{def}}{=} S_1(S_2\, q) \\
(S_1 \sqcup S_2)\, q &\overset{\mathrm{def}}{=} (S_1\, q) \cup (S_2\, q) \\
(S_1 \sqcap S_2)\, q &\overset{\mathrm{def}}{=} (S_1\, q) \cap (S_2\, q) \\
(\sqcap i \in I.\, S_i)\, q &\overset{\mathrm{def}}{=} (\cap i \in I.\, S_i\, q) \\
(\sqcup i \in I.\, S_i)\, q &\overset{\mathrm{def}}{=} (\cup i \in I.\, S_i\, q)\ .
\end{aligned}$$

1.2 Algebraic interpretation

The predicate transformers form a *category*, where the state spaces $\Sigma, \Gamma, \ldots$ are the *objects* of the category, and the *morphisms* $S : \Sigma \mapsto \Gamma$ are predicate transformers (we will write $\Sigma \mapsto \Gamma$ for the predicate transformers with initial state space Σ and final state space Γ). The identity morphisms in this category are the statement $\mathsf{skip} : \Sigma \mapsto \Sigma$ while sequential composition is the composition of morphisms.

The predicates on a state space Σ form the powerset of Σ, which is a *complete boolean lattice* with set inclusion as the ordering. The empty set (false) is the bottom of this lattice, while the universal set (true) is the top of the lattice. Intersection is lattice meet, union is lattice join, and complement (negation) is lattice complement.

The predicate transformers $\Sigma \mapsto \Gamma$ are ordered by the *refinement relation* (originally introduced by Back [1]), defined by

$$S \sqsubseteq S' \quad \overset{\mathrm{def}}{=} \quad (\forall\, q.\, S\, q \subseteq S'\, q).$$

The predicate transformers $\Sigma \mapsto \Gamma$ also form a complete boolean lattice with respect to the refinement relation. The statement abort is the bottom of this lattice, while magic is the top of the lattice. Meet is $S_1 \sqcap S_2$ and join is $S_1 \sqcup S_2$. Statement $(\sqcap i \in I.\, S_i)$ is the meet of the set of statements $\{S_i \mid i \in I\}$, while $(\sqcup i \in I.\, S_i)$ is the join of this set of statements. Both exist, because the lattice of predicate transformers is complete.

Finally, we can consider the statements $\{p\}$ and $[p]$ as two different *embeddings* of the lattice of predicates over Σ into the lattice $\Sigma \mapsto \Sigma$ of predicate transformers. Similarly, we can consider $\langle f \rangle$ as an embedding of functions $f : \Sigma \to \Gamma$ into predicate transformers $\langle f \rangle : \Sigma \mapsto \Gamma$, and $\{Q\}$ and $[Q]$ as two different embeddings of relations $Q : \Sigma \to \Gamma \to \mathsf{Bool}$ into predicate transformers $\Sigma \mapsto \Gamma$. One can show that these embeddings are in fact quite regular, in the sense that they preserve much of the lattice and category structure of the domain. For instance, we have that $\{p \cap q\} = \{p\} \sqcap \{q\}$, so that meets in the predicate lattice are mapped to meets in the predicate transformer lattice, and we have similarly that $\{p \cup q\} = \{p\} \sqcup \{q\}$. Also, we have $\{\mathsf{false}\} = \mathsf{abort}$, so the bottom of the predicate lattice is mapped to the bottom of the predicate transformer lattice. However, $\{\mathsf{true}\} \neq \mathsf{magic}$, so the top of the predicate lattice is not mapped to the top of the predicate transformer lattice. From a category theoretic view, we can show that all these embeddings are in fact *functors*, when state predicates, state transformers and state relations are viewed as categories of the appropriate kind.

For more detailed investigations of predicate transformers in the light of category theory we refer to Naumann [9] and Martin [7].

1.3 Monotonicity

A predicate transformer S is *monotonic*, if $p \subseteq q$ implies that $S\,p \subseteq S\,q$. The set of monotonic predicate transformers with initial state space Σ and final state space Γ form a complete (but not boolean) lattice, with the same top, bottom, meet and join as the lattice of all predicate transformers. Sequential composition is monotonic with respect to the refinement ordering, when we restrict ourselves to monotonic predicate transformers:

$$(S \sqsubseteq S') \sqcap (T \sqsubseteq T') \quad \Rightarrow \quad S;T \sqsubseteq S';T'. \qquad (*)$$

The monotonic predicate transformers form a subcategory of the predicate transformer category, with same identity element and same composition operation. This category is a *complete lattice enriched category*, because the morphisms (monotonic predicate transformers) from Σ to Γ form a complete lattice and the property $(*)$ holds.

The program statements are all monotonic, when interpreted as predicate transformers. On the other hand, we can show that any monotonic predicate transformers can be expressed as statements, so the statements form a *normal form* for the monotonic predicate transformers. In fact, any monotonic predicate transformer S can be expressed as a statement $\{P\};[Q]$, for some choice of relations P and Q (that depends on the predicate transformer S).

The program statements are thus built using basic algebraic constructs that have very regular mathematical properties. We can argue that the statement constructs are not introduced *ad hoc* but each one is a basic lattice theoretic or category theoretic construct, or justified by the embedding of predicates, state transformers and relations into the predicate transformers. The question now is whether these constructs also make sense operationally, i.e., do we have a good informal and operational interpretation of statements, so that the language can be used in the design of computing systems. Below we show that this is indeed the case.

2 Operational interpretation of statements

We consider a situation where a *user* interacts with a *computing system* through a common interface consisting of *global, shared variables*. Both the user and the system can read and write the global variables. A *program statement* describes the way in which the interaction between the user and system takes place. Essentially, it describes the interaction as a sequence of small incremental changes to the global state determined by the global variables. Below, we will go through the constructs of our language one by one, and explain informally how they would be executed.

2.1 Changing the state

The *skip statement* skip is a dummy statement, that does not change the state. Hence, it is very easy to execute. The *deterministic update* $\langle f\rangle$ computes a new state $f(\sigma)$ from the present state σ. The skip statement is a special case of the deterministic update where f is the identity function on the state, i.e., $\mathsf{skip} = \langle \mathsf{id}\rangle$.

An *assignment statement* like $x := x + 1$, which increments state component x by one, is a special kind of deterministic update, as is a *multiple assignment statement* like $x, y := x + y, x - y$, which assigns the sum of x and y to state component x and their difference to state component y. The assignment statement is expressed in terms of *program variables* x and y, that denote components of the state space, which we assume here to be a product of simpler types. We can use a *program variable declaration*, like

var x, y, z : Nat

to indicate that the state space has three components, which are called x, y, z, respectively, in this case all of type Nat. This declaration is a convenient shorthand that permits us to leave out the explicit lambda abstraction for state predicates, state transformations and state relations. The first assignment statement can then be expressed as $\langle x := x+1\rangle$ (or $\langle \lambda\, x, y, z.\ x := x+1\rangle$ if we use explicit lambda notation). The second assignment is expressed as the deterministic update $\langle x, y := x+y, x-y\rangle$ (or $\langle \lambda\, x, y, z.\ x, y := x+y, x-y\rangle$ if we again write out the lambda abstraction explicitly). The angular brackets are usually omitted around assignment statements like this, but are needed for updates with arbitrary state transformers.

2.2 Sequential composition

Sequential composition $S_1 ; S_2$ also has the usual meaning: it describes the order of progression in the computation. It permits us to describe a big computation as a sequence of smaller computation steps. Thus $\langle x := x+1\rangle ; \langle x, y := x+y, x-y\rangle$ carries out the two assignments in succession.

2.3 Failure

The *aborting* statement abort stands for a completely undefined computation. That is, we do not know what can happen if we execute this statement. The system may suddenly terminate in some inconsistent state, it might become completely unresponsive because it is stuck in an infinite loop, or any other nasty things may happen. It is also possible that the system just does some normal computation, but we do not know what. The main point is that we, as the user of the system, completely lose control of the computation. Executing an abort statement is thus a failure, and is something the user wants to avoid at all cost. Programs should therefore be designed in such a way that the abort statement is never executed.

The *assert* statement $\{g\}$ qualifies the aborting statement, so that aborting only occurs when condition g does not hold. For instance, the assertion $\{x \le y\}$ will abort the program execution in a state where $x \le y$ does not hold, but otherwise the statement has no effect. Abortion and skip are both special cases of the assert statement: $\{\mathsf{false}\} = \mathsf{abort}$ and $\{\mathsf{true}\} = \mathsf{skip}$ (remember that false is never satisfied while true is always satisfied in a state).

2.4 Selecting alternatives

The *angelic choice* statement $S_1 \sqcup S_2$ is interpreted as a request to the user to choose one of the two alternative statements. Execution is halted at this statement and the user is informed that he needs to make a choice. The user then has to indicate which of the statements S_1 or S_2 that he wants to continue with. Once he has done this, execution continues with the selected statement. Selections with more than two alternatives are defined by repeated selection, so that, e.g., $S_1 \sqcup S_2 \sqcup S_3 = (S_1 \sqcup S_2) \sqcup S_3$. Selection is associative, so it does not really matter how the expression is parenthesized.

The alternatives can be presented to the user in different ways. They can be given in a *menu*, the user may be asked for the number that identifies the selected alternative, or the alternatives may be associated with buttons on the screen. Other schemes for arranging the selection are also conceivable.

An angelic choice operator (the join operator for predicate transformer lattices) was introduced independently by Gardiner and Morgan [6] (as a "conjunction operator") and by Back and von Wright [2]. The operator was called "angelic" because the choice is made in such a way that the postcondition is established, if possible.

2.5 Input statement

The *angelic update* statement $\{Q\}$ also has a very familiar meaning in programs. It is a request for an input value. Executing this statement in state σ means that the system asks the user to choose a next state σ', from which to continue the execution. The choice must be such that relation $Q\,\sigma\,\sigma'$ holds. If there is no σ' that satisfies the condition, then the execution aborts.

An input statement will usually not ask the user to determine a completely new state, but only to select the value for some specific state component. An *angelic assignment* is an angelic update of the form $\{x := x' \mid Q\}$, where the user only needs to select a value for the state component(s) x, while all other state components retain their old values. For instance, a program to compute the square root could contain the statement $\{x, e := x_0, e_0 \mid x_0 \ge 0 \wedge e_0 > 0\}$, where x and e are two program variables. The statement asks the user to select a specific value $x_0 \ge 0$ for x and $e_0 > 0$ for

e (we will later use x as a value for which the square root is computed and e as the precision). The angelic assignment statement thus indicates which components should be given new input values, and also describes the legal inputs for these.

The assert statement is a special case of the angelic update where the state is not changed: $\{p\} = \{|p|\}$, where $|p|\,\sigma\,\sigma' \equiv (p\,\sigma \wedge \sigma = \sigma')$.

2.6 Waiting

The statement magic is known as a *miracle* [8]. The system cannot, of course, do a miracle, so it has to do what everybody else does: it has to wait for a miracle (which will never happen). In other words, the magic statement is really a *wait* statement. It can also be understood as a *deadlock* statement, because it simply prevents the execution from proceeding. A deadlock is not the same as an abortion, because it is not considered to be an error. On the contrary, it is an extreme safeguard against an error: rather than taking the risk of making an error, we do nothing.

The *guard* statement $[g]$ is a qualified wait statement, which will only wait when the condition g is not true. In this way, the guard g is a real guard: it permits execution to proceed when g holds, and forces the execution to wait when g does not hold. The guard can be understood as an *enabling condition*, which states under which condition it is permitted to execute the statement following the guard. For instance, the guard $[x < y]$ permits execution to proceed if $x < y$. The guard is mostly used in the composition $[g]; S$, which is abbreviated as $g \rightarrow S$, a so called *(naked) guarded command*. Miracle and skip are both special cases of the guard statement: $[\mathsf{false}] = \mathsf{magic}$ and $[\mathsf{true}] = \mathsf{skip}$.

2.7 Uncertainty

The *demonic choice* statement $S_1 \sqcap S_2$ reflects our uncertainty as to how the execution will proceed. We (the user of the program) do not know which one of the alternatives S_1 or S_2 will actually be chosen by the system. Since we do not have this information, we need to guard against either choice if we want to be certain to achieve some specific final state with our program (this justifies calling the choice “demonic”). Again, we can generalize the demonic choice to an arbitrary number of alternatives $S_1 \sqcap S_2 \sqcap \ldots \sqcap S_n$ by repeated choice.

Why would we, as program designers, want to permit this kind of uncertainty in our programs? Would it not be simpler to state explicitly which way the execution will proceed, and thus save ourselves the trouble of considering a number of different alternatives? In many situations we do not really have a choice, because we do not know which alternative will be chosen. In other situations, we do not really need to be completely explicit about what the program should do, it is sufficient that the statement behaves in a certain way for us to achieve the goals that we want to. It is better to leave ourselves some freedom to be more definite about the appropriate alternative later on, when we need to improve the program’s time or space efficiency, or make it better with respect to some other criteria.

The system is assumed to be eager to proceed, and will thus avoid waiting whenever possible. If one of the statements S_i in a demonic choice leads to a miracle and hence to a forced wait, then the system will not choose this one, if there is another choice that allows it to proceed. Consider as an example the statement

$$(x > 0 \rightarrow x := x + 1) \sqcap (x < 0 \rightarrow x := x - 1)$$

In an initial state where $x = 3$ the system would choose the first alternative, because the second alternative would lead to deadlock. In initial state $x = -3$, the second alternative would again be chosen. In initial state $x = 0$ the only choice is to wait.

2.8 Specification

The *demonic update* statement $[Q]$ (originally introduced by Back [1] as a nondeterministic assignment) describes a state change where the result may be uncertain. Executing $[Q]$ in an initial state σ, we know that execution will continue with some σ' such that $Q\,\sigma\,\sigma'$ holds, if possible, but we do not know exactly which state σ' will be selected, if there is more than one candidate. If there is no σ' such that $Q\,\sigma\,\sigma'$ holds, then the system must wait. The demonic update statement is a *specification* of what is actually done by the system during execution. An example is provided by a *demonic assignment* $[x := x_1 \mid -e \leq x - x_1^2 \leq e]$, which assigns to x some new value x_1 which is the square root of x with precision e.

Two successive executions of the same demonic update statement do not have to give the same final state. The reasons for this could be many. It could be that the actual implementation of the statement is based on some random natural phenomena and hence cannot be repeated. The implementation can be pseudo-random, in the sense that a random number generator is used to choose among the possible final states. Often the implementation has a hidden state that is preserved from one invocation of the statement to next, and this influences the choice the next time around. It is also possible that the same program is intended to run with different implementation versions of the specification and that these versions differ in some unimportant ways in how the new state is chosen. We are not allowed to know which version is actually being used, because the two versions should be mutually interchangeable.

The specification statement is a device by which we can control the information made available about program components. It allows us to postpone decision about details in the design until later. Only the information that is absolutely needed for successful use of a component should be given, and the rest is best hidden from the user. This gives the biggest possible freedom for the implementer of the component, to find a good implementation and to change the implementation when necessary, without affecting the other components in the program that use it. The demonic update statement is the basic construct for achieving this kind of information hiding.

3 Derived statements

Composing the basic statements in specific ways gives rise to a large selection of interesting and useful constructs. For instance, the statement

$$\begin{array}{l}\{x, e := x_0, e_0 \mid x_0 \geq 0 \wedge e_0 > 0\};\\ [x := x_1 \mid -e \leq x - x_1^2 \leq e]\end{array}$$

is a very general specification for a square root program: first the user is asked to select the value for which the square root is to be computed and the precision with which it is computed, and then the system computes an approximation of the square root with this precision.

Some statement constructs are so useful that it is motivated to give them a syntax of their own. We refer to these as *derived statements*. In practice, we usually work with the derived statements directly and can forget about how they were defined, provided that we have established enough useful properties of the derived statements. As a simple example of this, we will look at how to define conditional statements and while-loops as derived statements.

3.1 Conditional statements

The usual *guarded conditional* statement, introduced by Dijkstra [5], is defined by

$$\begin{array}{ll} & \mathsf{if}\ g_1 \rightarrow S_1 [\!] \ \ldots [\!]\ g_n \rightarrow S_n\ \mathsf{fi} \\ \overset{\mathrm{def}}{=} & \{g_1 \sqcup \ldots \sqcup g_n\}; ([g_1]; S_1 \sqcap \ldots \sqcap [g_n]; S_n).\end{array}$$

Thus, the conditional statement aborts if none of the guards g_i is satisfied. If at least one guard is satisfied, then one of the enabled statements is chosen for execution. The choice is demonic, so we do not know which one is chosen when two or more alternatives are enabled. However, as the system is eager to proceed, it will not choose an alternative that is not enabled, if there are enabled alternatives available. If the guards are mutually exclusive, the conditional statement is really deterministic, and there is no uncertainty associated with the outcome of the statement. This is the way conditional statements are usually defined in existing programming languages.

An example of a guarded command is

$$\mathsf{if}\ x \geq y \rightarrow x := y [\!]\ x \leq y \rightarrow \mathsf{skip}\ \mathsf{fi}$$

which sets x to the minimum of x and y.

A variant of the conditional statement would be the following statement

$$\begin{array}{ll}\mathsf{await}\, g_1 \to S_1 \,[\!]\ \ldots [\!]\ g_n \to S_n\ \mathsf{end} & \\ \overset{\text{def}}{=} \quad [g_1]; S_1 \sqcap \ldots \sqcap [g_n]; S_n . & \end{array}$$

This statement chooses to wait rather than abort, when none of the conditions is satisfied. In other words, the statements *awaits* the fulfillment of one of the enabling conditions. A construct like this could be used to synchronize the execution of parallel processes.

A further variant of the conditional statement is

$$\begin{array}{ll}\mathsf{choose}\, g_1 : S_1 \,[\!]\ \ldots [\!]\ g_n : S_n\ \mathsf{end} & \\ \overset{\text{def}}{=} \quad \{g_1\}; S_1 \sqcup \ldots \sqcup \{g_n\}; S_n . & \end{array}$$

In this case, the alternatives are given as selections. The assert statements g_i are evaluated in the current state, and those alternatives for which the conditions holds are presented to the user to choose from. The expectation is that one of the conditions is true (if not, then execution will abort). Thus, we have a context dependent selection. If the conditions are mutually exclusive, then the choice is deterministic, and there is no need to bother the user with making a selection. In this case, this construct and the guarded conditional statement have exactly the same behavior.

Finally, we have the variant

$$\begin{array}{ll}\mathsf{select}\, g_1 : S_1 \,[\!]\ \ldots [\!]\ g_n : S_n\ \mathsf{end} & \\ \overset{\text{def}}{=} \quad [g_1 \sqcup \ldots \sqcup g_n]; (\{g_1\}; S_1 \sqcup \ldots \sqcup \{g_n\}; S_n). & \end{array}$$

In this case execution may only proceed if we know that there is at least one alternative that can actually be selected. In other words, the system waits rather than aborts if no alternative is enabled.

3.2 Iteration and recursion

There are two statement constructs that we have not yet considered, the *general angelic choice* $(\sqcup i \in I \,.\, S_i)$ and the *general demonic choice* $(\sqcap i \in I \,.\, S_i)$. These are just the generalizations of the angelic and demonic choice statements so that the choice can be made among an arbitrary, even infinite, set of statements $\{S_i \mid i \in I\}$. In the special case that I is empty, the general angelic choice is defined to be abort and the general demonic choice is defined to be magic.

We will illustrate the angelic choice by defining the traditional *while loop* in terms of it. We define the while loop by

$$\mathsf{while}\ g\ \mathsf{do}\ S\ \mathsf{od} \overset{\text{def}}{=} (\sqcup i \in \mathsf{Nat} \,.\, \mathsf{Do}\, i),$$

where

$$\begin{array}{rcl}\mathsf{Do}\, 0 & = & \mathsf{abort} \\ \mathsf{Do}\,(i+1) & = & \mathsf{if}\ g \to S; \mathsf{Do}\, i \,[\!]\ \neg g \to \mathsf{skip}\ \mathsf{fi}, \quad i \geq 0.\end{array}$$

We assume here for simplicity that the statement S is deterministic, to avoid problems involving unbounded nondeterminism. In other words, Do 0 will abort immediately, Do will terminate normally if g holds, otherwise it will execute S and then it will abort, Do 2 will try one more iteration before aborting, and so on. In general, the statement Do $(i+1)$ will execute statement S at most i times. If the loop would terminate within i iterations, then Do $(i+1)$ will also terminate, with the same result. If more iterations would be required, then the statement Do $(i+1)$ will abort instead. Thus, Do i is the ith *approximation* of the loop.

The infinite angelic choice describes the execution of the loop. The user tries to avoid aborting, so in each state, he will choose an approximation Do i that is big enough so that the loop will terminate, if this is possible. If there is no such i, then the loop would also not terminate. The user is then forced to choose one of the aborting alternatives (which one does not matter, as the computation will fail anyway).

In programming languages, the iteration is usually implemented in a different way, but the effect is the same. Rather than letting the user make the choice, the system tries the approximations $\mathsf{Do}\,0, \mathsf{Do}\,1, \mathsf{Do}\,2, \ldots$ one by one, by at each stage iterating once more, if termination was not achieved with the previous choice. If there is a greatest number i of iterations required, then the effect will be exactly the same as if the user would have made the (right) choice $\mathsf{Do}\,(i+1)$ directly. If, however, the loop does not terminate, then the system will actually be stuck an infinite loop. Both situations are equally bad, and in the Refinement Calculus, these two possibilities are identified, because in both cases, the user has lost control over the computation.

Recursive statements can be defined in the same way, in terms of infinite angelic choice among approximation. The main point here is that we do not need to postulate an explicit iteration (or recursion) construct, because essentially the same computational behavior can be described with infinite angelic choice. We choose the latter as a basic statement constructor in our language, because it has much nicer mathematical properties than an explicit loop or recursion construct. The infinite demonic choice is also useful for describing programming notions, but we will not go into details of this here.

4 Correctness

Having introduced a notation for program statements, we next need to decide what to do with them. Basically, we use program statements to describe how computations are to be performed. A central question is then whether the statement that we have defined can be used for what we need, i.e., whether the statement is correct with respect to some given specification. The other central question is how to improve on the statement, in a way that preserves correctness, i.e., how to refine it.

Let us first consider what it means for a program statement to be correct. For statements that do not contain any choice constructs, the question is quite easy to answer. First, we *specify* what the program is supposed to do by giving a *precondition* p that may be assumed to hold for the initial state and a postcondition q that is required to hold for the final state. The program S is then *correct* with respect to this specification, if the final value computed by the program satisfies q whenever the initial value satisfies p. This is denoted $p \,\{\!|\; S \;|\!\}\, q$. As an example, we have that

$$x = 0 \,\{\!|\; \langle x := x+1 \rangle \;|\!\}\, x = 1.$$

In other words, if $x = 0$ initially, then the assignment statement will change the state so that $x = 1$.

Consider now what it means for S to be correct when the statement exhibits angelic nondeterminism. In this case, the user is given different choices for how to proceed during the computation. The statement S can be considered correct if the user can always make his choices so that q is established when S is started in an initial state that satisfies p. In other words, the user *can* establish postcondition q. An example is

$$x = 0 \,\{\!|\; \langle x := x+1 \rangle \sqcup \langle x := x+2 \rangle \;|\!\}\, x = 1.$$

In this case, the user can make his choice so that in the final state $x = 1$.

When our statement S exhibits demonic nondeterminism, then we require that q must be established by S when started in an initial state that satisfies p, no matter how the nondeterminism is resolved by the system during execution. In other words, the system *cannot avoid* establishing q. As an example, we have that

$$x = 0 \,\{\!|\; \langle x := x+1 \rangle \sqcap \langle x := x+2 \rangle \;|\!\}\, (x = 1 \vee x = 2).$$

The system must choose either statement, so it cannot avoid setting establishing the required postcondition. However, $x = 0 \,\{\!|\; \langle x := x+1 \rangle \sqcap \langle x := x+2 \rangle \;|\!\}\, x = 1$ does not hold, because the system might choose the second alternative.

The execution of a program can now be seen as a game between the user and the system. The rules of the game are given by the statement S as we have explained above, the possible initial positions of the game by the precondition p, and the winning positions for the user are determined by the postcondition q. Correctness of S with respect to precondition p and postcondition q, $p \,\{\!|\; S \;|\!\}\, q$, will then mean that the user has a *winning strategy* for reaching q from any initial state in p. In other words, the user can always win the game, no matter how the system plays [3].

5 Refinement of statements

Once we have a statement that is correct in the sense above, how can we make it better (more efficient, more portable, less space hungry etc.) without losing the correctness. In other words, how do we *refine* the program to better suit our purposes. Intuitively, program refinement means modifying a program while preserving correctness. This means that the refinement S' of S must be correct with respect to any pre-postcondition pair (p, q) for which S is correct. Thus, S is *refined by* S' if

$$(\forall\, p, q.\, p \,\{\!|\, S \,|\!\}\, q \Rightarrow p \,\{\!|\, S' \,|\!\}\, q) \qquad (*)$$

We can show that this condition is equivalent to $S \sqsubseteq S'$ as we defined it earlier. Hence, the refinement relation captures the notion of a correctness preserving refinement.

For angelic constructs, the notion of refinement that we have chosen implies that adding new alternatives to an angelic program is a refinement. Thus, $S_1 \sqcup S_2$ is refined by $S_1 \sqcup S_2 \sqcup S_3$, for any S_3. Given the intuitive explanation of angelic choice above, it should be clear that this does indeed preserve correctness. In the refined program, we will be offered more choices than we had in the original program. However, the old alternatives remain, and we may select these to achieve whatever we wanted to achieve with the original program. The refined program has then more *capabilities*, because there may be things we can achieve (final states that we can reach) that we could not achieve before. In this sense, refinement increases the power of the statement.

A particular case of refinement is when we replace abort with a statement that does not abort. In this case, the capabilities of the statement are also increased, because where earlier abortion was unavoidable, it may now be possible to continue execution and reach some interesting final state. This refinement could be interpreted as fixing a component that was broken.

Refining an assert statement means making the condition more permissive, so that less states will cause abortion. We are thus improving this component by making it work properly in situations where it would not work before.

An example of refinement of an angelic statement is

$$\begin{array}{ll} & \{x, e := x_0, e_0 \,|\, x_0 \geq 0 \wedge e_0 \geq 0.05\} \\ \sqsubseteq & \{x, e := x_0, e_0 \,|\, x_0 \geq 0 \wedge e_0 \geq 0.01\}. \end{array}$$

In this case, the user of the refined angelic assignment has more choices for the precision e than what he had in the original program.

Refinement of a demonic program statement is quite different from refinement of an angelic statement. Refinement for demonic statements means decreasing the uncertainty about the effect of executing the statement. A demonic choice $S_1 \sqcap S_2 \sqcap S_3$ is, e.g., refined by the choice $S_1 \sqcap S_2$. In this case, we have removed one possible option that the system could choose. If the program worked correctly no matter which of the alternatives S_1, S_2 or S_3 was selected by the system, it obviously has to work correctly if the system has less alternatives to choose from. We will know more about what could happen when the statement is executed. In other words, refinement means that the uncertainty is decreased and information is increased.

For a specification $[Q]$, refinement means that the choice of the next state is more deterministic. This usually means that the specification is closer to an actual implementation in a programming language. Program construction often proceeds in a stepwise manner from a very general specification to a very specific and deterministic implementation, intended to be a refinement of the original specification.

A deterministic statement S can in fact be refined even further. One possibility is to refine it to $g \rightarrow S$, i.e., associate a nontrivial enabledness condition with it. The statement S itself is equivalent to $\mathsf{true} \rightarrow S$, so refinement here just means tightening the conditions for enabledness. In this case, we consider the introduction of a deadlock as an improvement of the original program which did not deadlock. This might seem paradoxical, but it can be justified as follows: The statement S itself could lead to an abortion, and hence the whole execution would fail. However, the statement $g \rightarrow S$ might be disabled in the situation where S would abort. In this sense, we prefer waiting (delayed execution) to explicit failure.

An example of refinement of a demonic statement is the following:

$$[x := x_1 |\, -e \leq x - x_1^2 \leq e]$$

$$\sqsubseteq \quad [x := x_1 \,|\, -e/2 \leq x - x_1^2 \leq e/2].$$

In this case, the precision with which the square root is computed has been doubled, and hence, the user has more information about what the final value of x will be.

In general, we see that refinement of program statements amounts to improving the user's control of how the program is executed, either by decreasing the uncertainty of how a statement is actually executed, or by adding explicit new alternative ways for the user to choose. In the game theoretic interpretation, refinement means changing the program so that the user's possibilities of winning are increased (or at least, not decreased).

By the monotonicity of sequential composition, we can combine the above two refinements:

$$\begin{array}{ll} & \{x, e := x_0, e_0 \,|\, x_0 \geq 0 \wedge e_0 \geq 0.05\};\ [x := x_1 | -e \leq x - x_1^2 \leq e] \\ \sqsubseteq & \{x, e := x_0, e_0 \,|\, x_0 \geq 0 \wedge e_0 \geq 0.01\};\ [x := x_1 \,|\, -e/2 \leq x - x_1^2 \leq e/2]. \end{array}$$

The refined program gives both more choices to the user, and computes the square root with greater precision than the original program.

6 Conclusions

We have above described a simple language for describing the interaction between a user and a system. The basic constructs of the language are motivated by their fundamental algebraic properties. We gave a rather simple reading of the program constructs used in the refinement calculus, including such constructs as miracles, angelic choices and angelic updates, which have traditionally been considered quite exotic. We showed that a reasonable interpretation of such constructs is obtained by not only considering what the computing system can do, but also taken into account what the user of the system can do. This interpretation will also give a rather straightforward interpretation of correctness and refinement of programs with angelic and demonic nondeterminism. We will The themes treated in this paper are investigated in more detail in a book [4].

The results above can also be seen as an argument against the standard collection of fundamental statements in programming logics. Constructs like conditional statements and while loops, which are usually taken to be fundamental, are in fact much less regular than the basic lattice and category theoretic operations that we propose here. Similarly, the discussion above can be seen as a motivation for working in the full lattices of predicate transformers, rather than restricting oneself to weaker structures like complete partial orders.

References

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

[2] R.J. Back and J. von Wright. Duality in specification languages: a lattice-theoretical approach. *Acta Informatica*, 27:583–625, 1990.

[3] R.J. Back and J. von Wright. Games and winning strategies. *Information Processing Letters*, 53:165–172, 1995.

[4] R.J. Back and J. von Wright. *Refinement Calculus*. Book in preparation.

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

[6] P.H. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1):143–162, 1991.

[7] C.E. Martin. *Preordered Categories and Predicate Transformers*. PhD thesis, Oxford University, 1991.

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

[9] D.A. Naumann. *Two-Categories and Program Structure: Data Types, Refinement Calculi and Predicate Transformers*. PhD thesis, University of Texas at Austin, 1992.