Enriched Languages

Peter Baumann
Institut für Informatik der Universität Zürich
Winterthurerstr. 190, 8057 Zürich, Switzerland
Email: baumann@ifi.unizh.ch, Tel: ++41 1 257 45 79, Fax: ++41 1 363 00 35

Keywords: formal approaches, natural semantics, program verification

1 Introduction

Programmers are often faced with the problem of working in two languages: a specification language and a programming language. This is especially true when program development starts with a formal specification written in a language such as Z [BN+92] and ends with an imperative program. Over the years it has become clear that post facto program verification is not a practical approach. Therefore, it is well recognized that a sound and yet practical approach to developing imperative programs from formal specifications is needed. The main goal is a unified framework in which specification assertions and executable statements can be combined smoothly.

The purpose of this paper is to present a very general approach how to enrich imperative programming languages in the direction of wide spectrum languages. The key idea is to incorporate specification assertions into programming languages by allowing predicates on pairs of states to appear wherever a statement could appear. For example, a predicate $x' < x$ in Z[1] specifies an operation that decreases the value of the variable x. We regard it as a syntactically correct statement, although it is not executable. In the same way, `while` $x \geq 5$ `do` $(x' < x)$ is regarded as syntactically correct and we will give it a formal semantics in a natural semantics style. Intuitively, it describes a computation that decreases x until it is less then 5. Although it is in some sense more concrete than the statement $x \geq 5 \Rightarrow x' < x$, it is still not executable and can be further refined to `while` $x \geq 5$ `do` $x := x - 1$. This shows the general pattern of our approach. Starting with a specification, program development proceeds by fading out specification assertions and fading in executable statements until there are no more specification assertions. The point is that all statements belong to one language, which is given a formal semantics. This allows expressing precisely the relationship between the different statements.

The semantic basis of the method is natural semantics which allows omitting restrictions on the applicability of the method in case of interference. In contrast, aliasing prevents the use of many methods [Heh93, Mor88, Rey81]. Natural semantics are one of the very few formal semantics that are effectively used today to describe real world programming languages [MTH90]. Our approach of extending programming languages towards wide spectrum languages makes heavy use of the ability to express all constructs of the enlarged language in natural semantics.

[1]In Z, x' denotes the value of x after the operation.

2 Integrating Assertions into Programming Languages

A large and viable class of specification languages (e.g. Z, VDM [Jon90], etc.) in use today employs a model-based approach. In these languages, a specification describes a system as an abstract machine with states and operations which transform the states. Specifications of operations are build up from assertions on two states – the before and after states – and hence denote binary relations between states. Notational conventions help to distinguish between the variables for the before state and the ones for the after states. To keep our approach independent of a particular specification language, we do not concern ourselves with the detailed syntax of specifications, but assume that specifications are given by assertions P on pairs of states. We do not require that all functions appearing in assertions are total. A specification language may use lazy connectives, variants of Scott's-E logics or even three-valued logics. All we assume is that states $\sigma \in \Sigma$ are finite functions from variables to some values and that a satisfaction relation $(\sigma, \sigma') \models P$ is defined for pairs of states.

For illustrative purposes, we apply in our examples the notational convention of Z, which uses undecorated variables for the before states and primed variables for the after state. In these examples, states σ are taken from the set of finite functions $\Sigma = \mathit{Var} \rightarrow_{\mathit{fin}} \mathbb{N}$. An undecorated variable x for the before state σ has the meaning $\sigma(x)$, whereas the meaning of the corresponding x' for the after-state σ' is $\sigma'(x)$. Hence, $x' < x$ holds if the value of x in the after-state is less than its value in the before-state. An implicit assumption on assertions P is that state-variables remain unchanged if they are not mentioned.

To start, we illustrate our approach with a simple *while* language. The well-known rules for a natural semantics are given in Fig. 1, omitting the details for arithmetic and boolean expressions.

$$(\texttt{skip}, \sigma) \rightarrow \sigma \qquad \frac{(a, \sigma) \rightarrow n}{(x \;\texttt{:=}\; a, \sigma) \rightarrow \sigma[n/x]}$$

$$\frac{(b, \sigma) \rightarrow \texttt{true} \quad (c_1, \sigma) \rightarrow \sigma'}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma) \rightarrow \sigma'} \qquad \frac{(b, \sigma) \rightarrow \texttt{false} \quad (c_2, \sigma) \rightarrow \sigma'}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma) \rightarrow \sigma'}$$

$$\frac{(c_1, \sigma) \rightarrow \sigma'' \quad (c_2, \sigma'') \rightarrow \sigma'}{(c_1\ ;\ c_2, \sigma) \rightarrow \sigma'} \qquad \frac{(b, \sigma) \rightarrow \texttt{false}}{(\texttt{while}\ b\ \texttt{do}\ c, \sigma) \rightarrow \sigma}$$

$$\frac{(b, \sigma) \rightarrow \texttt{true} \quad (c, \sigma) \rightarrow \sigma'' \quad (\texttt{while}\ b\ \texttt{do}\ c, \sigma'') \rightarrow \sigma'}{(\texttt{while}\ b\ \texttt{do}\ c, \sigma) \rightarrow \sigma'}$$

Figure 1: Standard natural semantics of a *while* language

The questions is how this relational view of statements can be combined with specification assertions. At this simple level, there are at least two ways to unify assertions and programming language constructs:

1. Adapting a predicative programming view of programming language constructs:
   One can look at a statement c as a predicate on pairs of states and then take a natural semantics as a way of defining the semantics of c:

$$(\sigma, \sigma') \models c \text{ iff } (c, \sigma) \rightarrow \sigma'$$

   Accordingly, a rule like

$$\frac{(b, \sigma) \rightarrow \mathtt{true} \quad (c_1, \sigma) \rightarrow \sigma'}{(\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \sigma) \rightarrow \sigma'}$$

   could yield an inference rule

$$\frac{b \qquad c_1}{\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2}$$

2. Integrating specification assertions as (non-executable) statements into the programming language:
   On the syntax level, this requires extending the grammars and parsers for statements to include assertions. Type checking rules, if present, would have to be adapted as well. With regard to the dynamic semantics, the new class of statements has to be given a natural semantics. If we are working with a specification language that has no natural semantics of the form $(P, \sigma) \rightarrow \sigma'$, we can view assertions as basic statements with the following rule:

$$\frac{}{(P, \sigma) \rightarrow \sigma'} \ (\sigma, \sigma') \models P$$

We do not know whether and how the first approach would scale up to more complicated programming languages with block structures and procedures. The second possibility is the one we consider here. We will use meta-variables $p, q, \ldots$ to range over the expanded set of statements.

As an example, here is an excerpt of a proof that $(\mathtt{while}\ x \geq 5\ \mathtt{do}\ x' < x, \sigma) \rightarrow \sigma'$ with $\sigma(x) = 6$ and $\sigma'(x) = 3$:

$$\frac{\dfrac{\vdots}{(x \geq 5, \sigma) \rightarrow \mathtt{true}} \quad \dfrac{}{(x' < x, \sigma) \rightarrow \sigma'}\ (\sigma, \sigma') \models (x' < x) \quad \dfrac{\dfrac{\vdots}{(x \geq 5, \sigma') \rightarrow \mathtt{false}}}{(\mathtt{while}\ x \geq 5\ \mathtt{do}\ x' < x, \sigma') \rightarrow \sigma'}}{(\mathtt{while}\ x \geq 5\ \mathtt{do}\ x' < x, \sigma) \rightarrow \sigma'}$$

In general, the enriched language allows nondeterminism by the introduction of assertions, but well-known laws about program equivalences still hold. This seems to imply that we do not have to change the semantics of the original programming language constructs. This point will be further investigated in the next section. Another issue, addressed in the fourth section, is how one obtains executable programs from programs containing non-executable assertions. Finally, in the fifth section, we show how our approach scales up to more complicated languages.

3 Angelic and Demonic Interpretations in Natural Semantics

Let us have a closer look at the following example program $x' < x;\ x := x - 1$. Here is a proof that $(x' < x;\ x := x - 1, \sigma) \to \sigma[0/x]$ for a state σ with $\sigma(x) = 2$:

$$\dfrac{\dfrac{}{(x' < x, \sigma) \to \sigma[1/x]} \quad (\sigma, \sigma[1/x]) \models x' < x \quad \dfrac{\dfrac{\vdots}{(x - 1, \sigma[1/x]) \to 0}}{(x := x - 1, \sigma[1/x]) \to \sigma[0/x]}}{(x' < x;\ x := x - 1, \sigma) \to \sigma[0/x]}$$

If we choose to decrease x by 2 in the first statement, the expression $x - 1$ in the second statement becomes undefined. This is problematic since we lose compositionality of program development: although $x := x - 2$ can replace $x' < x$, the composition $x := x - 2;\ x := x - 1$ cannot replace $x' < x;\ x := x - 1$.

The point is that the standard rule for sequential composition gives too optimistic a view in case of nondeterminism: if there is one intermediate state on which the second statement terminates, the composite statement can terminate. People working on formal program development call such an interpretation *angelic*. But in software development, one should adapt a pessimistic view: whenever a program can fail to terminate it will. Therefore, we need a *demonic* interpretation of sequential composition. The usual approach is to allow termination only if the second statement terminates on all possible intermediate states. This can also be captured in natural semantics by attaching a side condition to the standard rule:

$$\frac{(p_1, \sigma) \to \sigma'' \quad (p_2, \sigma'') \to \sigma'}{(p_1\ ;\ p_2, \sigma) \to \sigma'} \qquad \forall\, \overline{\sigma} \bullet (p_1, \sigma) \to \overline{\sigma} \Rightarrow \exists\, \overline{\overline{\sigma}} \bullet (p_2, \overline{\sigma}) \to \overline{\overline{\sigma}}$$

With this rule, there is no σ' such that $(x' < x;\ x := x - 1, \sigma) \to \sigma'$ for σ with $\sigma(x) = 2$.

Similar reflections show that the rule for the while-construct has to be changed, too. In order to get a succinct demonic interpretation, we need some auxiliary notions. The relation $\succ_{p,b} \subseteq \Sigma \times \Sigma$ consists of the pairs (σ, σ') such that $((p, \sigma) \to \sigma' \wedge (b, \sigma) \to \mathtt{true})$. A *tracegraph* $tg(\sigma, p, b) = (V, \succ_{p,b}|_V)$[2] for statement p and condition b with root state σ is the directed graph with $V = \{\sigma' \in \Sigma \mid \sigma \overset{*}{\succ}_{p,b} \sigma'\}$, where $\overset{*}{\succ}_{p,b}$ denotes the reflexive and transitive closure of $\succ_{p,b}$. A tracegraph $tg(\sigma, p, b)$ represents all the possible computation sequences of p starting in σ as long as b is true. A tracegraph $tg(\sigma, p, b)$ is *regular* if $\succ_{p,b}|_V$ is well-founded and all the minimal elements σ' of V have the property that $(b, \sigma') \to \mathtt{false}$. A regular tracegraph $tg(\sigma, p, b)$ captures the idea that all possible computation sequences are finite and terminate such that $\neg\, b$ holds. The following important property of tracegraphs holds:

$(\sigma, \sigma') \in \overset{*}{\succ}_{p,b}$ implies
  $tg(\sigma', p, b) \subseteq tg(\sigma, p, b)$ and
  $tg(\sigma, p, b)$ *regular* implies $tg(\sigma', p, b)$ *regular*.

The rules for the demonic interpretation of `while` are:

[2] $\succ_{p,b}|_V$ is the restriction of $\succ_{p,b}$ to V.

$$\frac{(b,\sigma) \rightarrow \texttt{false}}{(\texttt{while}\ b\ \texttt{do}\ p, \sigma) \rightarrow \sigma}$$

$$\frac{(b,\sigma) \rightarrow \texttt{true} \quad (p,\sigma) \rightarrow \sigma'' \quad (\texttt{while}\ b\ \texttt{do}\ p, \sigma'') \rightarrow \sigma'}{(\texttt{while}\ b\ \texttt{do}\ p, \sigma) \rightarrow \sigma'}\ tg(\sigma, p, b)\ \mathit{regular}$$

In the following, we use the demonic interpretations of sequential composition and of the while-construct. In the case where we have deterministic assertions or statements the angelic and demonic semantics coincide. It is also worth noting that the familiar laws about program equivalences still hold.

The example with the sequential composition shows that it becomes important to know on which set of states a statement or a program is defined. This motivates the introduction of the notion of a precondition of a statement p, written $\mathit{pre}\ p$. Intuitively, $\mathit{pre}\ p$ is true in a state σ, if there exists a state σ' such that $(p, \sigma) \rightarrow \sigma'$. Most importantly, the semantics of preconditions can be defined in natural semantics style, too. Clearly, we can add a rule like

$$\frac{(p,\sigma) \rightarrow \sigma'}{(\mathit{pre}\ p, \sigma) \rightarrow\ \mathit{true}}$$

This is the rule we add for assertions to keep independence of a particular specification language. For executable statements, some of the proofs become simpler if we add the rules of Fig. 2.

$$(\mathit{pre}\ \texttt{skip}, \sigma) \rightarrow \texttt{true} \qquad \frac{(a,\sigma) \rightarrow n}{(\mathit{pre}\ (x\ :=\ a), \sigma) \rightarrow \texttt{true}}$$

$$\frac{(b,\sigma) \rightarrow \texttt{true} \quad (\mathit{pre}\ p_1, \sigma) \rightarrow \texttt{true}}{(\mathit{pre}\ (\texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2), \sigma) \rightarrow \texttt{true}} \qquad \frac{(b,\sigma) \rightarrow \texttt{false} \quad (\mathit{pre}\ p_2, \sigma) \rightarrow \texttt{true}}{(\mathit{pre}\ (\texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2), \sigma) \rightarrow \texttt{true}}$$

$$\frac{(\mathit{pre}\ p_1, \sigma) \rightarrow \texttt{true}}{(\mathit{pre}\ (p_1\ ;\ p_2), \sigma) \rightarrow \texttt{true}}\ \forall \overline{\sigma} \bullet (p_1, \sigma) \rightarrow \overline{\sigma} \Rightarrow (\mathit{pre}\ p_2, \overline{\sigma}) \rightarrow \texttt{true}$$

$$\frac{}{(\mathit{pre}\ (\texttt{while}\ b\ \texttt{do}\ p), \sigma) \rightarrow \texttt{true}}\ tg(\sigma, p, b)\ \mathit{regular}$$

Figure 2: Preconditions for the *while* language

The last construct we want to introduce in this section is the choice operator $\sqcup$. A first attempt could be

$$\frac{(p_1,\sigma) \rightarrow \sigma'}{(p_1 \sqcup p_2, \sigma) \rightarrow \sigma'} \qquad \frac{(p_2,\sigma) \rightarrow \sigma'}{(p_1 \sqcup p_2, \sigma) \rightarrow \sigma'}$$

But this is again an angelic interpretation. The demonic interpretation in natural semantics style is given by:

$$\frac{(pre\ p_1,\sigma)\rightarrow \mathtt{true} \quad (p_2,\sigma)\rightarrow\sigma'}{(p_1\sqcup p_2,\sigma)\rightarrow\sigma'} \qquad \frac{(pre\ p_2,\sigma)\rightarrow \mathtt{true} \quad (p_1,\sigma)\rightarrow\sigma'}{(p_1\sqcup p_2,\sigma)\rightarrow\sigma'}$$

and the corresponding rule for the precondition is:

$$\frac{(pre\ p_1,\sigma)\rightarrow \mathtt{true} \quad (pre\ p_2,\sigma)\rightarrow \mathtt{true}}{(pre\ (p_1\sqcup p_2),\sigma)\rightarrow \mathtt{true}}$$

4 Relating Statements in the Enriched Language

Having accepted that some of our programs contain non-executable statements, we have to provide a means to relate them to executable programs. Therefore, what we are looking for is a formal refinement relation $\sqsubseteq$ between the constructs of the enriched language. This relation should express the fact that we can safely replace one statement by another statement.

A standard approach is to define $p \sqsubseteq q$ by:

for all σ, σ'

1. $(pre\ p,\sigma)\rightarrow \mathtt{true}$ implies $(pre\ q,\sigma)\rightarrow \mathtt{true}$ and
2. $(pre\ p,\sigma)\rightarrow \mathtt{true}$ and $(q,\sigma)\rightarrow\sigma'$ implies $(p,\sigma)\rightarrow\sigma'$

The first obligation captures the requirement that q should at least provide results in all circumstances where p is defined. The second obligation says that the results of q in the situations where p is defined should fulfill the requirements of the results of p.

It is easy to see that $p \sqsubseteq q$ is equivalent to: for all σ,σ' $(p\sqcup q,\sigma)\rightarrow\sigma'$ iff $(p,\sigma)\rightarrow\sigma'$. In other words, q is considered a refinement of p iff the statements $p\sqcup q$ and p are semantically equivalent, and hence $p\sqsubseteq q$ and $q\sqsubseteq p$ implies that p and q are semantically equivalent.

Based on this definition of $\sqsubseteq$ and the natural semantics, we can start to provide sufficient conditions for typical pairs of constructs to be in $\sqsubseteq$. It is worth noting that some of the conditions are given at the semantic level. If one is working with a concrete specification language one might be able to turn them into rules on the syntactic level.

Assignment introduction rule

$$p \ \sqsubseteq\ x := a$$

if, for all σ, $(pre\ p,\sigma)\rightarrow \mathtt{true}$ implies $(a,\sigma)\rightarrow \mathrm{n}$ and $(p,\sigma)\rightarrow\sigma[n/x]$.

Soundness
The soundness of this rule is immediate. □

An example application is:

$$(x' < x) \sqsubseteq x := x - 1$$

But it can also be applied between pure programs

$$x := x+1;\ x := x+1 \sqsubseteq x := x+2$$

A similar rule can be given for the skip-construct. But a more interesting rule is the one for sequential composition.

Composition introduction rule

$$p \sqsubseteq p_1 \;;\; p_2$$

iff for all σ, σ' and σ''

- $(pre\ p, \sigma) \rightarrow \mathtt{true}$ implies $(pre\ p_1, \sigma) \rightarrow \mathtt{true}$
- $(pre\ p, \sigma) \rightarrow \mathtt{true}$ and $(p_1, \sigma) \rightarrow \sigma''$ implies $(pre\ p_2, \sigma'') \rightarrow \mathtt{true}$
- $(pre\ p, \sigma) \rightarrow \mathtt{true}$ and $(p_1, \sigma) \rightarrow \sigma''$ and $(p_2, \sigma'') \rightarrow \sigma'$ implies $(p, \sigma) \rightarrow \sigma'$

Soundness
We first show that these conditions are sufficient to discharge for all states σ, σ'

1. $(pre\ p, \sigma) \rightarrow \mathtt{true}$ implies $(pre\ (p_1 \;;\; p_2), \sigma) \rightarrow \mathtt{true}$
2. $(pre\ p, \sigma) \rightarrow \mathtt{true}$ and $(p_1 \;;\; p_2, \sigma) \rightarrow \sigma'$ implies $(p, \sigma) \rightarrow \sigma'$

The proof of the first obligation follows from the observation that $pre\ (p_1 \;;\; p_2, \sigma) \rightarrow \mathtt{true}$ is equivalent to $(pre\ p_1, \sigma) \rightarrow \mathtt{true}$ and $\forall\, \overline{\sigma} \bullet ((p_1, \sigma) \rightarrow \overline{\sigma} \Rightarrow (pre\ p_2, \overline{\sigma}) \rightarrow \mathtt{true})$. The truth of the second obligation is obvious from the third assumption.

To show that the above conditions are also necessary is equally simple. □

IfThenElse introduction rule

$$p \sqsubseteq \mathtt{if}\ b\ \mathtt{then}\ p_1\ \mathtt{else}\ p_2$$

if $p \wedge b \sqsubseteq p_1$ and $p \wedge \neg\, b \sqsubseteq p_2$

Soundness
Soundness is shown by simple case analysis. □

While introduction rule

$$p \sqsubseteq \mathtt{while}\ b\ \mathtt{do}\ inv \wedge q$$

if for all σ and σ'

- $(pre\ p, \sigma) \rightarrow \mathtt{true}$ implies $tg(\sigma, inv \wedge q, b)$ *regular*
- $(pre\ p, \sigma) \rightarrow \mathtt{true}$ and $(\mathtt{while}\ b\ \mathtt{do}\ inv \wedge q, \sigma) \rightarrow \sigma'$ implies $(inv, \sigma) \rightarrow \sigma'$
- $(pre\ p, \sigma) \rightarrow \mathtt{true}$ and $(inv, \sigma) \rightarrow \sigma'$ and $(b, \sigma') \rightarrow \mathtt{false}$ implies $(p, \sigma) \rightarrow \sigma'$

Soundness
The soundness of this rule is based on the fact that $(\mathtt{while}\ b\ \mathtt{do}\ inv \wedge q, \sigma) \rightarrow \sigma'$ implies $(b, \sigma') \rightarrow \mathtt{false}$. The invariant used is an assertion on pairs of states. □

In order to discharge these conditions, one normally uses the induction principles associated with natural semantics. Good overviews can be found in [Gun92, Win93].

This set of basic refinement rules can be extended to include other cases, especially for connectives used in a particular specification language. But apart from the refinement rules, it is a necessary condition for the successful application of our approach that the programming operators are monotonic with respect to the refinement relation.

Theorem 1
$p \sqsubseteq q$ implies

- $p\ ;\ r \sqsubseteq q\ ;\ r$ *and* $r\ ;\ p \sqsubseteq r\ ;\ q$
- $p \sqcup r \sqsubseteq q \sqcup r$ *and* $r \sqcup p \sqsubseteq r \sqcup q$
- $\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ r \sqsubseteq \texttt{if}\ b\ \texttt{then}\ q\ \texttt{else}\ r$ *and*
  $\texttt{if}\ b\ \texttt{then}\ r\ \texttt{else}\ p \sqsubseteq \texttt{if}\ b\ \texttt{then}\ r\ \texttt{else}\ q$
- $\texttt{while}\ b\ \texttt{do}\ p \sqsubseteq \texttt{while}\ b\ \texttt{do}\ q$

Proof sketch
The proofs of the conjectures for sequential composition, choice and if-then-else are straightforward. The proof for the while-construct is a bit more involved. The first step is to show that $tg(\sigma, p, b)$ *regular* and $p \sqsubseteq q$ imply $tg(\sigma, q, b) \subseteq tg(\sigma, p, b)$, and hence $tg(\sigma, q, b)$ *regular*. The second step is to use case analysis on b and well-founded induction. □

The following theorem contains another indispensable property of the refinement relation, which is easy to prove.

Theorem 2
$p \sqsubseteq q$ and $q \sqsubseteq r$ implies $p \sqsubseteq r$

To conclude this section, here is an example development of the computation of the quotient and remainder :

$$
\begin{array}{ll}
x = q' \times y + x' \wedge y > 0 \wedge x' < y \wedge q = 0 & \\
\sqsubseteq & (\textit{While introduction rule}) \\
& \texttt{while}\ x \geq y\ \texttt{do}\ (q \times y + x = q' \times y + x') \wedge x' < x \\
\sqsubseteq & (\textit{Refinement on assertions}) \\
& \texttt{while}\ x \geq y\ \texttt{do}\ x' = x - y \wedge q' = q + 1 \\
\sqsubseteq & (\textit{Compostion and assignment introduction rules}) \\
& \texttt{while}\ x \geq y\ \texttt{do}\ x := x - y\ ;\ q := q + 1
\end{array}
$$

5 Extension to More Complicated Languages

In the second section, we presented our approach with the help of a simple programming language. More complicated languages have semantic judgements of the form $A \vdash B \rightarrow C$, where A normally stands for an environment keeping track of the scopes of program variables. It is common in natural semantics to describe the dynamic semantics of a block structured language including procedures and functions with the help of environments and stores instead of states only. Environments *env* are usually finite maps from the set of program variables *Var* to a set *Loc*, representing the set of possible memory locations. Stores ρ are finite maps from *Loc* to the set of possible values. In order to retrieve the value of a variable x in the environment *env* for store ρ, the composition $(\rho \circ env)$ is applied to x. It is this splitting of states into environments and stores that gives natural semantics much of its power. In particular, it becomes possible to easily describe different procedure-calling mechanisms [Plo81].

With these notions, the judgements for the dynamic semantics of imperative languages have the form

$$env \vdash (c, \rho) \rightarrow \rho'$$

For example, the new assignment rule is:

$$\frac{env \vdash (a, \rho) \rightarrow n}{env \vdash (x \ := \ a, \rho) \rightarrow \rho[n/l]} \quad env(x) = l$$

Declarations can be given a formal semantics as well by judgements of the form

$$env \vdash (d, \rho) \rightarrow (env', \rho')$$

saying that a declaration d alters the current environment and possibly the store. A possible rule for a variable declaration where dv stands for a default value and where new is a function returning an unused location is

$$\frac{}{env \vdash (\texttt{var} \ \ z, \rho) \rightarrow (env[l/z], \rho[dv/l])} \quad l = new(env)$$

A block usually has a rule similar to

$$\frac{env \vdash (d, \rho) \rightarrow (env'', \rho'') \quad env'' \vdash (c, \rho'') \rightarrow \rho'}{env \vdash (\texttt{begin} \ \ d, \ \ c \ \ \texttt{end}, \rho) \rightarrow \rho' \mid_{-env''(DV(d))}}$$

where $DV(d)$ represents the set of variables declared in d and where $\rho' \mid_{-env''(DV(d))}$ is equal to ρ' except that it is not defined for the locations of the new variables in $DV(d)$.

In contrast, specification languages do not use notions like environments or stores in the above sense. Therefore, we still assume that we have a specification language with assertions on pairs of states. A specifier must not care about environments and stores!

In order to make our approach also work in this situation, we need a new way to bring the two worlds together. The main problem here is that the same state can be described by different environments and stores. As a consequence, a statement like $x := y + 1$ is a correct implementation of a specification $x' \neq y'$ unless x and y interfere in the environment. Therefore, the correctness of implementations depends on the chosen environment. This is known as the interference problem [Rey81].

Coming back to our approach of integrating assertions as statements into the target programming language, we can adapt the main idea of section 2 to the more complicated situation here by defining:

$$env \vdash (P, \rho) \rightarrow \rho' \quad \text{ iff } \quad (\rho \circ env, \rho' \circ env) \models P$$

for assertions P.

All our notions and definitions now contain a new environment component, on which their validity depends. Preconditions are defined as follows:

$$env \vdash (pre \ p, \rho) \rightarrow \texttt{true} \text{ iff there exists } \rho' \text{ such that } env \vdash (p, \rho) \rightarrow \rho'$$

As an example, here is the precondition rule for the block structure:

$$\frac{env \vdash (d,\rho) \rightarrow (env'',\rho'') \quad env'' \vdash (pre\ p,\rho'') \rightarrow \mathtt{true}}{env \vdash (pre\ (\mathtt{begin}\ \ d,\ \ p\ \ \mathtt{end}),\rho) \rightarrow \mathtt{true}}$$

The validity of the refinement relation depends on the environment, too. We define

$$env \vdash p \sqsubseteq q$$

iff for all ρ, ρ'

1. $env \vdash (pre\ p,\rho) \rightarrow \mathtt{true}$ implies $env \vdash (pre\ q,\rho) \rightarrow \mathtt{true}$
2. $env \vdash (pre\ p,\rho) \rightarrow \mathtt{true}$ and $env \vdash (q,\rho) \rightarrow \rho'$ implies $env \vdash (p,\rho) \rightarrow \rho'$

With these definitions, all the rules of sections 2 and 3 can be adapted accordingly. Here is the corresponding assignment introduction rule

$$env \vdash p \sqsubseteq\ x := a$$

if for all ρ, $env \vdash (pre\ p,\rho) \rightarrow \mathtt{true}$ implies $env \vdash (a,\rho) \rightarrow \mathtt{n}$ and $env \vdash (p,\rho) \rightarrow \rho[n/env(x)]$.

An example of a new refinement rule which can be proven valid for all environments env is:

Local variable introduction rule

$$env \vdash p \sqsubseteq \mathtt{begin\ var}\ \ z,\ \ p \wedge z' = z'\ \ \mathtt{end}$$

if z does not occur in p.

Furthermore, monotonicity holds

$$env'' \vdash p \sqsubseteq q \text{ implies } env \vdash \mathtt{begin\ var}\ \ z,\ \ p\ \ \mathtt{end} \sqsubseteq \mathtt{begin\ var}\ \ z,\ \ q\ \ \mathtt{end}$$

where $env'' = env[new(env)/z]$.

More interesting examples are procedures and their calling mechanisms. Let us have a look at call-by-reference as found in PASCAL. This is interesting because the formal and actual parameters interfere, and many development methods can not handle such situations [Heh93, Mor90, Rey81]. The semantic rule for procedure declarations is (see [Plo81] for details):

$$env \vdash (\mathtt{proc}\ proc(form\_paras)\ \ c,\rho) \rightarrow (env[(\lambda\, locs.env[locs/form\_paras], c)/proc],\rho)$$

and the rule for procedure calls is:

$$\frac{env' \vdash (second(env(proc)),\rho) \rightarrow \rho'}{env \vdash (proc(act\_paras),\rho) \rightarrow \rho'}\ env' = first(env(proc))(env(act\_paras))$$

In the above rule, *first* and *second* are projections.

Procedure introduction rule (call-by-reference)

$$env \vdash p \sqsubseteq proc(act\_paras)$$

if the formal parameters *form_paras* do not occur in p and if $env' \vdash p \sqsubseteq c$, where $c = second(env(proc))$ and $env' = first(env(proc))(env(act\_paras))$.

As an example, we take the following procedure *swap*:

```
proc swap(x, y)
begin
    var z,
        z := x; x := y; y := z
end
```

Let *env* be such that $env(a) \neq env(b)$ and $env(swap) = (\lambda(l_1, l_2).env[l_1/x, l_2/y], c)$, where c represents the body of *swap*. Then we have $env'(x) = env(a)$ and $env'(y) = env(b)$ for $env' = first(env(swap))(env(a), env(b))$. Let $a' = b \wedge b' = a$ be the top-level specification. Then:

$$\begin{array}{ll} env' \vdash a' = b \wedge b' = a & \sqsubseteq \texttt{begin var } z,\ a' = b \wedge b' = a \texttt{ end} \\ & \sqsubseteq \texttt{begin var } z,\ x' = y \wedge y' = x \texttt{ end} \\ & \sqsubseteq \texttt{begin var } z,\ z := x;\ x := y;\ y := z \texttt{ end} \\ \text{implies} \quad env \vdash a' = b \wedge b' = a & \sqsubseteq swap(a, b) \end{array}$$

In general, we start program development with a top-level environment which is injective on the free variables of the top-level specification.

6 Summary

We have shown how to enrich imperative languages in the direction of wide spectrum languages by:

1. incorporating specification assertions on pairs of states as statements
2. adapting a demonic interpretation of the constructs and defining their preconditions
3. defining a refinement relation between statements of the expanded language

Most importantly, this can be done on a sound semantic basis provided by a natural semantics for the enriched language. To the best of our knowledge, this is the first approach to incorporate assertions on pairs of states into programming languages, thereby reducing the gap between specification languages and programming languages.

Another innovative idea is to base this incorporation on a natural semantics. This allows omitting restrictions on the applicability of the method in the presence of interference. Many approaches on formal program development use variants of axiomatic semantics [Gri81, Rey81, Jon90]. Our idea is similar to the one in the Refinement Calculus [Mor88], where a special specification statement based on pre- and postconditions is added to Dijkstra's guarded command language. But the Refinement Calculus is based on a weakest-precondition semantics and cannot handle interference [Mor88]. Our method is also different from predicative programming [Heh93] in that these approaches try to integrate programming constructs as predicates into specification logics, but again with the restriction that no aliasing may occur. The inclusion of predicates into an extended language of Standard ML is investigated in research on Extended ML using a natural semantics. But this done for the pure functional subset of SML only [KST93].

The possibility of connecting our approach with the model-oriented specification language Z has been described in [Bau94]. The goal is to develop a concrete wide spectrum languages based on the ideas presented in this paper.

Acknowledgement

The author would like to thank Ina Kraan for her helpful comments on earlier versions of this paper.

References

[Bau94] P. Baumann. Z and natural semantics. In J. Bowen and A. Hall, editors, *Proceedings of the 1994 Z User Meeting*, pages 168–184. Springer-Verlag, 1994.

[BN+92] S. Brien, J. Nicholls, et al. Z Base Standard version 1.0. Oxford University, 1992.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages, Structure and Techniques.* MIT Press, 1992.

[Heh93] Eric C.R. Hehner. *A Practical Theory of Programming.* Springer-Verlag, 1993.

[Jon90] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, 2 edition, 1990.

[KST93] St. Kahrs, D. Sannella, and A. Tarlecki. The semantics of Extended ML: A gentle introduction. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Langauges*, pages 186–215. Springer-Verlag, 1993.

[Mor88] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[Mor90] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Dep. of Computer Science, Aarhus University, September 1981.

[Rey81] John C. Reynolds. *The Craft of Programming.* Prentice-Hall, 1981.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.