T U M

I N S T I T U T F Ü R I N F O R M A T I K

A Brief Study in Automating Proofs Based on a Refined Hoare-logic

Peter Müller
Arnd Poetzsch-Heffter

TUM-I9635
November 96

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-11-96-I9635-200/1.-FI

Druck: Fakultät für Mathematik und
Institut für Informatik der
Technischen Universität München

A Brief Study in Automating Proofs Based on a Refined Hoare-logic

Peter Müller[1]
Arnd Poetzsch-Heffter[2]

Institut für Informatik
Technische Universität
München

November 12, 1996

[1]muellerp@informatik.tu-muenchen.de
[2]poetzsch@informatik.tu-muenchen.de

Abstract

This report deals with program verification based on a refined Hoare-logic which allows to handle procedure calls.

A certain specification technique allows to specify these procedures by pre- and postconditions. To do that, the data model of the programming language is formalized and objects of the programming language are mapped to abstract values. Specifications can thus refer to these abstract values and describe the behavior of a procedure on a higher level of abstraction.

As basic operations of the programming language can cause exceptions they are considered as procedures. This allows to specify their behavior and prove the absence of certain exceptions. The disadvantage of this approach is that procedure calls play an even more prominent role in verification.

Handling procedures makes automation of correctness proofs much harder because it is not possible to compute the weakest precondition of a procedure call in most cases. Two examples show how suitable preconditions can be found.

On the one hand, the enhancement of the programming logic enables our framework to deal with realistic programs by handling e. g. side-effects or recursion. On the other hand it leads to larger and more complex correctness proofs. It is shown that this additional effort can mostly be done by a verification system.

1 Introduction

Increasing use of computer systems in safety-critical areas leads to a strong demand for extremely reliable software. This level of reliability can only be achieved by means of formal methods, in particular by correctness proofs (cf. [Hoa96]).

Previous work (cf. [PHB96]) has shown that carrying out correctness proofs — even for very small programs — tends to become very strenuous. Therefore a high grade of automation is required to make proving more efficient and economically reasonable.

Examination of correctness proofs has shown that most proof steps are straightforward. This leads to the notion of interactive program verification where most parts of the proof are carried out automatically by a computer. Human interaction is required in cases where intuition is needed to solve more difficult problems.

The whole work is embedded in the Lopex research project[3]. It is concerned with tools for the support of formal methods in program development. In particular, it deals with so-called logic based programming environments which allow the specification and verification of object-oriented programs.

2 Specification

In our framework specifications are split into two interacting parts: A program-independent part which contains e. g. the specification of abstract data types and a program-dependent part which links parts of the program to parts of the abstract specification.

The program-independent part is formalized by the specification language of a proof checker like e. g. PVS (cf. [PVS95]). This allows to prove program-independent lemmata automatically.

Program-dependent properties are specified by so-called annotations as known from [Hoa69]. These are formulae of the predicate calculus. Annotations can occur as pre- or postconditions of procedures or as class invariants. To keep things simple we neglect class invariants in this report and concentrate on imperative programs rather than on object-oriented ones.

To build the link between the world of objects and the world of abstract specifications, our specification technique has to support data abstraction. Therefore we use so-called abstraction functions to map objects or whole object structures to terms of abstract data types. This allows to specify the behavior of procedures on a higher level of abstraction.

Basically, our specification technique is very similar to Larch's two-tiered approach (cf. [GH93]). As we aim to program verification we are forced to give

[3] http://wwweickel.informatik.tu-muenchen.de/forschung/lopex/lopex_e.html

our specifications precise semantics and overcome some of Larch's shortcomings (cf. [CGR96]). A more detailed comparison can be found in [PH96].

To illustrate our technique we look at a small example. Suppose that the abstraction function for integer objects is denoted by $\bar{}$ and that *range* is a unary predicate that holds if its argument lies within the boundaries of the integer representation of the programming language.

$\bar{} : int \rightarrow Integer$
$range : Integer \rightarrow Boolean$

In most cases specifications of procedures describe a relation between the parameters passed to the procedure and its return value. As statements in the body of the procedure are allowed to change the values of the parameters it is not sufficient to describe the result in terms of the parameters or, in general, in terms of program variables. Therefore we introduce logical variables[4] in the precondition which denote the abstractions of the parameters' values. Changes to the parameters do not have any influence on these logical variables. They still contain the abstractions of the parameters' initial values.

Now we can specify a procedure `times` that takes two arguments and yields their product. If the abstractions of the parameters are denoted by the logical variables A and B, we have to assure that $A * B$ does not exceed the limits of integer representation. In each case the precondition meets this demand we guarantee that the abstraction of the result, which is denoted by the variable *result*, yields $A * B$.

`int times (int n, int m)`
`pre` $\overline{n} = A \wedge \overline{m} = B \wedge range(A * B)$
`post` $\overline{result} = A * B$

Applying `times` in a correct way (i. e. in states where the precondition is fulfilled) assures that the multiplication is carried out without the risk of raising an arithmetic exception.

3 Verification

The fact that [Hoa69] is one of the most widely cited papers in computer science shows that Hoare-logic is considered to be the best fit verification technique known by now.

In this report we use parts of the Hoare-style logic that was developed by A. Poetzsch-Heffter to prove the partial correctness of object-oriented programs. The reader may refer to [PH96] for a detailed discussion and Appendix A for an overview of the rules of our calculus.

[4] cf. section 3 for the discrimination of logical and program variables

Most interesting features of our logic are (a) the discrimination between logical and program variables and (b) how procedure calls are treated. (a) is a solution to a shortcoming of Hoare logic. As logical and program variables behave different they are discerned in our framework. This allows to handle the rules for substitution and quantification more easily. (b) allows to incorporate the specification of a procedure into correctness proofs. This is done by the call-rule, the elim-rule and the inv-rule which can be found in Appendix A.

In order to make it possible to show the absence of most kinds of exceptions, we give the logic a stronger semantics. In pure Hoare-logic a triple $\{P\}$ S $\{Q\}$ means: If P holds in the state before the execution of S, Q will hold in the state after execution if S terminates regularly (cf. [Hoa69]). In our framework the semantics is: If P holds in the state before the execution of S, Q will hold in the state after execution except if S runs forever or causes a memory exception. Treating memory exceptions would require to include assumptions about the hardware und software environment in which the program is carried out. For simplicity these aspects of specification are not addressed in this paper.

The possibility of exceptions during the computation of expressions shows that most operators of a programming language don't have pure functional behavior. Therefore it is straightforward to consider those operations as procedure calls because they behave like procedures rather than like functions. These procedures can be specified as shown in section 2.

From this point of view, complex expessions are simply nested procedure calls. They can be reasoned about by splitting them up into single calls (cf. [Cou90]). This allows to incorporate specifications for all basic operations into correctness proofs and thereby prevent exceptions.

4 Automation of correctness proofs

The basic notion of proving programs correct is to deduce for each procedure `p` the triple $\{Precondition_p\}$ $Body_p$ $\{Postcondition_p\}$ in the programming logic. As Dijkstra and Gries showed in [Dij76] and [Gri81] this can be done by the so-called predicate transformer $wp(S, R)$ which yields the weakest precondition so that $\{wp(S, Q)\}$ S $\{Q\}$ holds.

Using wp, correctness proofs are carried out backwards, starting with the last statement of a block. The predicate transformer allows to step back through the block until the first statement is reached. This means that proving that $\{P\}$ S $\{Q\}$ holds is just the same as showing that $P \Rightarrow wp(S, Q)$.

Carrying out correctness proofs in practice shows that wp is not capable to deal with the real interesting parts of the proofs. Finding preconditions for assignments and `if` statements is quite helpful, but what is really needed is a method to handle iteration and procedure calls because they form the core of most programs.

As pointed out below, preconditions of `while` statements and procedure calls depend on annotations. Thus it is not possible to find the weakest precondition in general. The best guess that can be done is to figure out the weakest precondition relative to the specification of a `while` statement or procedure. We call this precondition a *suitable precondition.*

`while` statement In case of a `while` statement — as can be seen from the while-rule (App. A) — the suitable precondition has to meet two demands:

- It has to be an invariant of the loop.
- It must be strong enough to allow the deduction of the desired postcondition.

The weakest precondition suggested in [Gri81] is not practically applicable. It requires the computation of recursivly defined formulae. Each of these formulae contains the weakest precondition of the loop's body for a different postcondition. If the body encloses any procedure calls this is too strenuous. So we assume that a suitable precondition is given as annotation or entered by the user of a verification system.

Procedure call Finding the suitable precondition of a procedure call is more complicated. In top-down software development programs are composed of procedures that are not yet implemented. As we want to allow this development style, we have to ensure that verification can be done without knowing the code of such subordinate procedures. I. e. we have to rely completely on their specifications. Again, finding the weakest precondition is, in general, not possible (cf. [ZHL96]).

Searching for ways to find out a suitable precondition we first focus on procedures which don't cause any side-effects. In this case we can assume that postconditions always have the form $\overline{result} = E$, where E does not contain any program variables. A suitable precondition sp of a call of such a side-effect-free precedure `p` is given below.

$$sp(v := p(E_1, \ldots, E_n), Q) \equiv PRE_p[E_1/p_1 \ldots E_n/p_n] \wedge Q[E/\overline{v}]$$

PRE_p denotes the precondition of `p` in its specification. The formula above can be deduced as follows:

$$\dfrac{\dfrac{\dfrac{\{PRE_p\}\ \text{proc p}(p_1,\ldots,p_n)\ \{\overline{result}=E\}}{\{PRE_p[E_1/p_1\ldots E_n/p_n]\}\ \text{v} := \text{p}(E_1,\ldots,E_n)\ \{\overline{v}=E\}}\text{call-rule}}{\{PRE_p[E_1/p_1\ldots E_n/p_n]\wedge Q[E/\overline{v}]\}\ \text{v} := \text{p}(E_1,\ldots,E_n)\ \{\overline{v}=E\wedge Q[E/\overline{v}]\},\quad \overline{v}=E\wedge Q[E/\overline{v}]\Rightarrow Q}\text{inv-rule}}{\{PRE_p[E_1/p_1\ldots E_n/p_n]\wedge Q[E/\overline{v}]\}\ \text{v} := \text{p}(E_1,\ldots,E_n)\ \{Q\}}\text{weak-rule}$$

□

To use this precondition in practice, you have to get rid of those free logical variables that occur in PRE_p but not in Q. Most of these appear in an equation $V = E$ which makes it possible to replace all occurrences of V by E and discard the equation.

The remaining free variables can be bound by using the ex-rule (cf. App. A). This rule may be applied because the variable to be bound does not appear in Q. Now we have a precondition that contains not more unbound variables than the postcondition.

5 Example without side-effects

To illustrate the techniques described above we'll now introduce a small example program which computes the factorial of a given number iteratively. We will specify the properties and prove them.

To reason about a program it is necessary to have a formalisation of the data model of the programming language it is written in. As our program only deals with `int`-values a very small data model suffices. We use *Integer* to denote the infinite set of integer numbers while `int` stands for the integer representation of the programming language.

First we introduce constants to describe the bounadries of integer representation:

$Integer\ MIN = -2^{15}$
$Integer\ MAX = 2^{15} - 1$

We introduce a predicate $range(x)$ that holds if its argument lies within these boundaries:

$range : Integer \rightarrow Boolean$
$range(x) \equiv MIN \leq x \leq MAX$

Now we can specify the type `int`:

$int = \{x | range(x)\}$

We introduce an abstraction function to map `int` objects to $Integers$. As `int` is a subset of $Integer$ this is not necessary in this case but it shows the general use of abstraction functions to fill the gap between the domain of objects and the formal specifications.

$\bar{} : int \rightarrow Integer$

For the specification of our program a definition of the factorial function is required:

$! : Integer \rightarrow Integer$
$$n! = \begin{cases} 1 & : \quad n = 0 \\ n * (n-1)! & : \quad n > 0 \end{cases}$$

Our implementation of the factorial will be based on the arithmetic operators `minus` and `times` which are specified as follows:

`int times (int n, int m)`
`pre` $\overline{m} = A \wedge \overline{n} = B \wedge range(A * B)$
`post` $\overline{result} = A * B$

`int minus (int n, int m)`
`pre` $\overline{m} = A \wedge \overline{n} = B \wedge range(A - B)$
`post` $\overline{result} = A - B$

Both operations are side-effect-free and stick to the assumptions we made above. We can now take a close look to our implementation:

`int fac (int n)`
`pre` $\overline{n} = N \wedge 0 \leq \overline{n} \leq 12$
`post` $\overline{result} = N!$

```
{
int r := 1;
while (n > 1) {
    r := times(r, n);
    n := minus(n, 1);
    }
return r;
}
```

The specification states that whenever the argument `n` lies between 0 and 12, the procedure will yield `n` factorial if it terminates. It guarantees that no errors except memory errors will occur, in particular arithmetic overflow is ruled out by limiting `n` to a maximum value of 12. Otherwise it would be impossible to carry out a correctness proof.

The proof obligation we have to fulfill is to deduce the tripel $\{PRE_{fac}\}$ $BODY_{fac}$ $\{POST_{fac}\}$. We will show this by simulating a mechanical proof system. The first step of such a system would be to consider the program fragment in question. In our case it's a sequence of three statements. As we want to step through the program backwards, we break up this sequence between `while` and `return`. As we can see from the seq-rule, we have to find the precondition for the `return` statement. We do this by using wp.

$wp(\texttt{return r}, \overline{result} = N!) \equiv \overline{r} = N!$

So we can split our triple and get two new ones, of which the second can immediately be deduced from the return-rule:

$\{\overline{n} = N \wedge 0 \leq \overline{n} \leq 12\}$
```
int r := 1;
while (n > 1) {
    r := times(r, n);
    n := minus(n, 1);
    }
```
$\{\overline{r} = N!\}$

and

$\{\overline{r} = N!\}$
```
return r;
```
$\{\overline{result} = N!\}$

The program part of the first triple is, again, a sequence. This time we need the suitable precondition of the `while` statement which is entered by the user of our proof system.

In this case the appropriate invariant is $0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!$. Again, splitting up the triple delivers two new ones:

$\{\overline{n} = N \wedge 0 \leq \overline{n} \leq 12\}$
```
int r := 1;
```
$\{0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!\}$

and

$\{0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!\}$
```
while (n > 1) {
    r := times(r, n);
    n := minus(n, 1);
    }
```
$\{\overline{r} = N!\}$

The first triple can be deduced by applying the the assign-axiom and the weak-rule.

$wp(\texttt{r := 1}, 0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!) \equiv$
$0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{1} = N!/\overline{n}!$

We now have to show that

$\overline{n} = N \wedge 0 \leq \overline{n} \leq 12$ implies $0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{1} = N!/\overline{n}!$

which is obviously true and could be done by a proof checker. This completes the deduction of the first triple so that we can switch to the second one.

As the current triple consists of a `while` statement, we have to transform it in a way that makes it possible to use the while-rule. Therefore we have to weaken the postcondition. Is is quite easy to see (and can thus be shown by a mechanical prover) that

$\neg(n > 1) \wedge 0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!$ implies $\overline{r} = N!$

because $\overline{n}$ has to be 0 or 1. The new triple matches the while-rule:

$\{0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!\}$
```
while (n > 1) {
    r := times(r, n);
    n := minus(n, 1);
    }
```
$\{\neg(n > 1) \wedge 0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!\}$

Applying the rule results in a new triple concerned with the body of the loop:

$\{n > 1 \wedge 0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!\}$
```
r := times(r, n);
n := minus(n, 1);
```
$\{0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!\}$

Now we can apply the strategy for the treatment of procedure calls pointed out above. In a first step we compute

$sp(\texttt{n := minus(n, 1)}, 0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!)$,

which results in

$\overline{n} = A \wedge \overline{1} = B \wedge range(A - B) \wedge 0 \leq A - B \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/(A - B)!$

Eliminating unbound variables delivers

$range(\overline{n} - 1) \wedge 0 \leq \overline{n} - 1 \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/(\overline{n} - 1)!$

This gives us two new triples:

$\{n > 1 \wedge 0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!\}$

```
r := times(r, n);
```

$\{range(\overline{n}-1) \wedge 0 \leq \overline{n}-1 \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/(\overline{n}-1)!\}$

and

$\{range(\overline{n}-1) \wedge 0 \leq \overline{n}-1 \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/(\overline{n}-1)!\}$

```
n := minus(n, 1);
```

$\{0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!\}$

The deduction of a triple of the second kind was shown on page 5. We repeat the same steps for the first triple and receive as weakest precondition:

$$\overline{r} = A \wedge \overline{n} = B \wedge range(A * B) \wedge range(\overline{n}-1) \wedge$$
$$\wedge 0 \leq \overline{n}-1 \leq 12 \wedge 0 \leq N \leq 12 \wedge A * B = N!/(\overline{n}-1)!$$

Again, we can eliminate unbound variables which results in

$$range(\overline{r} * \overline{n}) \wedge range(\overline{n}-1) \wedge 0 \leq \overline{n}-1 \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} * \overline{n} = N!/(\overline{n}-1)!$$

What remains to show is that

$$n > 1 \wedge 0 \leq \overline{n} \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} = N!/\overline{n}!$$

implies

$$range(\overline{r} * \overline{n}) \wedge range(\overline{n}-1) \wedge 0 \leq \overline{n}-1 \leq 12 \wedge 0 \leq N \leq 12 \wedge \overline{r} * \overline{n} = N!/(\overline{n}-1)! \ ,$$

which can easily be done by a proof checker like the PVS system.

This completes the verification of our example program. The whole work could be done automatically except the step where the loop invariant is needed.

6 The Treatment of Side-Effects

As we want to deal with realistic programs, we can't be satisfied by limiting to side-effect-free procedures. The treatment of side-effects requires a much more elaborated specification and verification technique.

Side-effects are changes on the global state of program execution. They can be caused by the manipulation of global variables or, in object-oriented programming languages, by attribute updates or the creation of new objects. Side-effects can be handled by making the execution state explicit. Therefore we introduce $ to denote the current state of program execution. Side-effects can thus be described by specifying the changes made on $.

In a more sophisticated framework we can't any more stick to our assumption that postconditions always have the form $\overline{result} = E$ because we want to specify many different aspects of a procedure's behaviour, e. g. functional behaviour, side-effects, invariant clauses or the relation to other objects. To keep specifications modular und easily tractable we allow to specify different properties in seperate pairs of pre- and postconditions (cf. [PH95]).

As a consequence of this technique, the computation of a suitable precondition as described above doesn't work any more. Suppose we want to find a precondition P which allows to deduce $\{P\}$ `v := p(`$E_1, \ldots, E_n$`)` $\{Q\}$. P should be as weak as possible. The pre- and postcondition of the ith pre-post-pair of `p`'s specification are denoted by PRE_i and $POST_i$.

To deal with the new situation a mechanical verification system could either take the conjunction of all pre-post-pairs which would allow to deduce the strongest possible postcondition. In turn the precondition would also get very strong and is thereby maybe unprovable which makes this technique unusable.

The alternative way is to figure out which of the pairs are needed to deduce the desired postcondition Q. To do this, the system has to split Q into two parts Q_{proc} and Q_{inv} such that $Q_{proc} \wedge Q_{inv} \Rightarrow Q$. Q_{proc} is established by the procedure `p` and Q_{inv} must hold before the call of `p` and stay invariant during execution of `p`. Then a minimal set S of indices has to be computed such that $\bigwedge_{i \in S} POST_i \Rightarrow Q_{proc}$. Q can be reassembled by application of the inv-rule (cf. App. A). As first-order logic is not decidable a machine can neither find the decomposition of Q nor compute the set S.

This shows that, in general, proofs in our logic can't be carried out completely automated. Therefore our proof system must enable the user to interact and influence the proof. Automation can be increased by finding heuristics how to select the parts of a specification that are needed in a certain proof step. Therefore pattern matching algorithms may be applied to compare the desired postcondition Q with the available pre-post-pairs.

7 Example with side-effects

To study the treatment of side-effects we consider an example which makes use of global variables. Suppose we have a global variable `db` of type `int` and a procedure `set` to change its value. `set` behaves as follows: If its argument is greater or equal to 0, it is assigned to `db`. Otherwise `db` is left unchanged. In both cases the argument is returned. The specification would look like this:

`global int db;`

`int set(int n)`

`pre` $\overline{n} = N \wedge N \geq 0$
`post` $\overline{db} = N$

`pre` $\overline{n} = N \wedge N < 0 \wedge \overline{db} = DB$
`post` $\overline{db} = DB$

`pre` $\overline{n} = N$
`post` $\overline{result} = N$

We now want to find a precondition P that allows to deduce

$\{P\}$ `v := set(e)` $\{\overline{db} = \overline{e}\}$,

where `v` must not be a global variable (especially `v` $\neq$ `db`). Our system will examine each pair of pre- and postconditions and analyse its relevance for the next proof step.

First pair Matching the postcondition of the first pair against the desired postcondition makes clear that we have to establish $\overline{db} = N \wedge N = \overline{e}$ which implies $\overline{db} = \overline{e}$. Thus we receive $Q_{proc} \equiv \overline{db} = N$ and $Q_{inv} \equiv N = \overline{e}$. This results in precondition $P_1 \equiv \overline{e} = N \wedge N \geq 0$. which can be simplified to $\overline{e} \geq 0$.

Second pair We can carry out similar steps for the second pair and determine that $Q_{proc} \equiv \overline{db} = DB$ and $Q_{inv} \equiv DB = \overline{e}$. This implies that $P_2 \equiv \overline{e} = N \wedge N < 0 \wedge \overline{db} = DB \wedge DB = \overline{e}$. Eliminating free variables results in $\overline{e} < 0 \wedge \overline{db} = \overline{e}$.

Third pair As `v` does not appear in the considered formula, we can't match *result* against any term. Thus the third pair does not contribute to our goal.

Finding P Our examination of the specification has shown that there are two possibilities for P namely P_1 and P_2. As we are interested in a precondition that is as weak as possible, we deduce $P \equiv P_1 \vee P_2$.

8 Conclusion

We have shown how we can avoid all exceptions but memory errors by considering expressions as procedure calls. The examples demonstrate that although procedure calls can't be handled by a verification system in all cases, heuristics allow the treatment of many of them and thereby ease the work of proving. In particular, a machine can cope with procedures that don't have very complex specifications. In daily practice this is the most common case. Procedures implementing basic arithmetic and logic operations have very simple specifications even if their properties are specified more entirely than in section 5.

In other words, assuming we have a verification system as described above, the refined Hoare-logic is as easy to handle as pure Hoare-logic although it is a lot more powerful.

A Programming Logic

assign-rule: $\vdash \{\, \mathbf{P}[\mathbf{E}/\mathbf{v}] \,\}\ \ \mathrm{v} := \mathrm{E}\ \ \{\, \mathbf{P} \,\}$

return-rule: $\vdash \{\, \mathbf{P} \,\}\ \ \text{return v}\ \ \{\, \mathbf{P}[\mathbf{result}/\mathbf{v}] \,\}$

while-rule: $$\frac{\vdash \{\, \overline{\mathbf{EXP}} \wedge \mathbf{P} \,\}\ \ \mathrm{STAT}\ \ \{\, \mathbf{P} \,\}}{\vdash \{\, \mathbf{P} \,\}\ \ \text{while (EXP) \{ STAT \}}\ \ \{\, \neg\overline{\mathbf{EXP}} \wedge \mathbf{P} \,\}}$$

seq-rule: $$\frac{\vdash \{\, \mathbf{P} \,\}\ \ \mathrm{STAT1}\ \ \{\, \mathbf{Q} \,\}\ ,\ \ \vdash \{\, \mathbf{Q} \,\}\ \ \mathrm{STAT2}\ \ \{\, \mathbf{R} \,\}}{\vdash \{\, \mathbf{P} \,\}\ \ \mathrm{STAT1}\ ;\ \mathrm{STAT2}\ \ \{\, \mathbf{R} \,\}}$$

call-rule: $$\frac{\vdash \{\, \mathbf{P} \,\}\ \ \mathrm{proc}\ p(p_1, \ldots, p_n)\ \ \{\, \mathbf{Q} \,\}}{\vdash \{\, \mathbf{P}[\mathbf{E_1}/\mathbf{p_1}, \ldots, \mathbf{E_n}/\mathbf{p_n}] \,\}\ \ \mathrm{v} := \mathrm{p}(E_1, \ldots, E_n)\ \ \{\, \mathbf{Q}[\mathbf{v}/\mathbf{result}] \,\}}$$

rec-rule: $$\frac{\{\, \mathbf{P} \,\}\ \ \mathrm{proc}\ p(p_1, \ldots, p_n)\ \ \{\, \mathbf{Q} \,\} \vdash\ \{\, \mathbf{P} \,\}\ \ \mathrm{BODY}(\mathrm{proc}\ p)\ \ \{\, \mathbf{Q} \,\}}{\{\, \mathbf{P} \,\}\ \ \mathrm{proc}\ p(p_1, \ldots, p_n)\ \ \{\, \mathbf{Q} \,\}}$$

inv-rule: $$\frac{\vdash \{\, \mathbf{P} \,\}\ \ \mathrm{v} := \mathrm{p}(E_1, \ldots, E_n)\ \ \{\, \mathbf{Q} \,\}}{\vdash \{\, \mathbf{P} \wedge \mathbf{R} \,\}\ \ \mathrm{v} := \mathrm{p}(E_1, \ldots, E_n)\ \ \{\, \mathbf{Q} \wedge \mathbf{R} \,\}}$$

subst-rule: $$\frac{\vdash \{\, \mathbf{P} \,\}\ \ \mathrm{PART}\ \ \{\, \mathbf{Q} \,\}}{\vdash \{\, \mathbf{P}[\mathbf{t}/\mathbf{X}] \,\}\ \ \mathrm{PART}\ \ \{\, \mathbf{Q}[\mathbf{t}/\mathbf{X}] \,\}}$$

ex-rule: $$\frac{\vdash \{\, \mathbf{P} \,\}\ \ \mathrm{STAT}\ \ \{\, \mathbf{Q}[\mathbf{Y}/\mathbf{X}] \,\}}{\vdash \{\, \exists \mathbf{X} : \mathbf{P} \,\}\ \ \mathrm{STAT}\ \ \{\, \mathbf{Q}[\mathbf{Y}/\mathbf{X}] \,\}}$$

strength-rule: $$\frac{\mathbf{P} \Rightarrow \mathbf{Q}\ ,\ \ \vdash \{\, \mathbf{Q} \,\}\ \ \mathrm{STAT}\ \ \{\, \mathbf{R} \,\}}{\vdash \{\, \mathbf{P} \,\}\ \ \mathrm{STAT}\ \ \{\, \mathbf{R} \,\}}$$

weak-rule: $$\frac{\mathbf{R} \Rightarrow \mathbf{Q}\ ,\ \ \vdash \{\, \mathbf{P} \,\}\ \ \mathrm{STAT}\ \ \{\, \mathbf{R} \,\}}{\vdash \{\, \mathbf{P} \,\}\ \ \mathrm{STAT}\ \ \{\, \mathbf{Q} \,\}}$$

References

[CGR96] Patrice Chalin, Peter Grogono, and T. Radhakrishnan. Identification of and solutions to shortcomings of LCL, a larch/c interface specification language. In Marie-Claude Gaudel and James Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 385–404. Springer-Verlag, January 1996.

[Cou90] Patrick Cousot. Methods and logics for proving programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 15, pages 841–993. Elsevier Science Publishers B. V., 1990.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[GH93] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

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

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[Hoa96] C. A. R. Hoare. How did software get so reliable without proof? In Marie-Claude Gaudel and James Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, January 1996.

[PH95] A. Poetzsch-Heffter. Interface specifications for program modules supporting selective updates and sharing and their use in correctness proofs. In G. Snelting, editor, *Softwaretechnik 95*, 1995.

[PH96] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. *Habilitation thesis*, 1996. (to appear).

[PHB96] A. Poetzsch-Heffter and B. Bauer. Verification of class-based programs. (to appear), 1996.

[PVS95] *A Tutorial Introduction to PVS*, April 1995.

[ZHL96] Job Zwiers, Ulrich Hannemann, and Yassine Lakhneche. Modular completeness: Integrating the reuse of specified software in top-down program development. In Marie-Claude Gaudel and James Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 595–608. Springer-Verlag, January 1996.