Theorem Prover Support for Precondition and Correctness Calculation

Orieta Celiku
Joakim von Wright

Turku Centre for Computer Science

TUCS Technical Reports

No 449, March 2002

Theorem Prover Support for Precondition and Correctness Calculation

Orieta Celiku
Joakim von Wright

Turku Centre for Computer Science
TUCS Technical Report No 449
March 2002
ISBN 952-12-0972-0
ISSN 1239-1891

Abstract

Tools for automatically extracting the conditions for which a program is correct with respect to a precondition and postcondition can make proving program correctness easier. We build a HOL-based tool that uses weakest preconditions to prove correctness theorems with the verification conditions as assumptions. The programming language includes loops, (recursive) procedures, and also both demonic and angelic nondeterminism, which can be used to model interaction. Program variables can be of arbitrary types. Programs with procedures are handled modularly, and proved facts about individual procedures are stored in a database available to all programs. We also discuss two new rules for calculating loop preconditions and recursion correctness while taking into consideration specification variables.

TUCS Laboratory
Learning and Reasoning Laboratory

1 Introduction

Hoare logic and Dijkstra's weakest precondition are suitable for reasoning about program correctness but the proofs require a lot of work even for very small programs.

Calculating program correctness amounts to analyzing the program structure and collecting, for each statement, the verification conditions under which the statement is correct with respect to a given or calculated precondition and postcondition. Automation requires that sequences are annotated with intermediate assertions, loops are annotated with invariants and termination functions, and that procedures are annotated with preconditions, postconditions and if recursive also with invariants and termination functions.

It is desirable that the proof method of programs with procedures reflects the modular way such programs are designed, developed and used [5]. Modularity of proofs can be achieved by first proving that the body of a procedure satisfies some specification and then using this result to derive conclusions about a program that employs the procedure.

This paper describes a tool for calculating preconditions and (total) correctness, based on the HOL theorem prover [7]. The tool uses weakest preconditions to automatically extract the verification conditions for the correctness of suitably annotated programs. The programming language includes nondeterministic statements (both angelic and demonic), loops, and procedures with value and reference parameters, where procedures can also be recursive. Programs with procedures are handled in a modular way, and a database mechanism allows for procedures to be employed even from programs that do not declare them. The language embedding is shallow, and program variables can have any HOL type.

The paper also contributes two rules for calculating preconditions of loops and correctness of recursive procedures. The rules are new in the way they take the presence of specification variables into account. Moreover, the loop rule makes it possible to treat loops modularly.

An automatic correctness calculator can lessen the burden of proving programs correct. Such a tool could be integrated into a software development tool and used, if not to verify full programs, to prove correctness of interesting or problematic pieces of programs. By adding angelic nondeterminism to the language we have widened the range of programs that can be modeled and analyzed to include interactive and game-like systems [1].

2 Related work

Verification condition generators and the embedding of different programming languages into theorem provers have been going in and out of fashion for a while. Here we concentrate on what we found most relevant to our work; for a detailed overview see Homeier [8].

Gordon's work [6] on a shallow embedding of a deterministic while-language (as well as other constructs from dynamic logic, etc.) in the HOL system has been an inspiration for many other embeddings. Hoare-logic rules are used to prove correctness and a HOL tactic generates verification conditions for programs with annotations. Annotations include assertions for sequences of statements when the second statement is not an assignment.

Homeier [8] performs a deep embedding of a deterministic while-language with procedures in HOL, where expressions can also have side effects. A method for proving correctness of recursive procedures, including mutually recursive ones, is implemented. Hoare-logic rules are used to automatically generate verification conditions and the verification condition generator is proved sound. A significant limitation of this work is that the programs are assumed to work on an initial state where the program variables are assigned explicit values; hence it is impossible to prove correctness assertions for arbitrary initial states. Another limitation is that program variables can only be of natural number type.

Nipkow [13] deeply embeds a small deterministic while-language in the Isabelle proof assistant and describes a verification condition generator built for this language. Partial correctness is analyzed by calculating the weakest liberal precondition of the statements while generating the corresponding verification conditions. The verification condition generator is proved sound and relatively complete. However, the focus is on embedding several semantics for the language following a program semantics textbook, so the language is quite basic, variables are only of natural number type, and there is no treatment of termination or mentioning of specification variables.

Filliatre [3] formalizes a language with both functional and imperative features in the Coq system. The language has recursive functional procedures but not mutually recursive ones. Program variables are variables of the logic and hence can have any type. A Coq tactic for proving program correctness is implemented. The correctness proof starts with the construction of an incomplete proof term whose skeleton is the program. The gaps in the proof term are later filled with the corresponding verification conditions. Loop and recursion termination are established by proving that the variant decreases over a wellfounded relation. The need for explicit auxiliary variables is eradicated by use of labels, for example $a@0$ gives the value of

a before the statement labeled by 0. A nice package of sorting examples is verified.

3 The correctness/precondition calculator

Our tool builds on a previous formalization of a programming language [2] in the HOL system. Programs are modeled as predicate transformers (functions from state predicates to state predicates) according to a weakest precondition semantics.

We analyze program correctness by calculating weakest preconditions. A total correctness assertion $p\{\!|c|\!\}q$ is equivalent to $p \subseteq c\ q$, where $\subseteq$ means "stronger than" and $c\ q$ is the weakest precondition for statement c to establish postcondition q. An advantage of taking this approach to program correctness is that instead of inventing an intermediate assertion for the sequence statement we can automatically use as intermediate assertion the weakest precondition of the second statement with respect to the postcondition.

3.1 Language embedding

The programming language is shallowly embedded in the HOL logic, which means that the terms in the embedded language are identified with terms in the HOL logic by extralogical parsing. The shallow embedding allows the identification of types in the embedded language with types in the HOL logic. Hence, when types are mentioned they refer to HOL types.

The program state is implemented as a tuple where every component corresponds to the value of a program variable. The state has a polymorphic type Σ that for each program is instantiated to the product of the concrete types of the program variables. For example, the state for a program with three variables $x : \mathsf{num}$ and $y : \mathsf{bool}$ and $z : \mathsf{real}$ is of type $\mathsf{num} \times \mathsf{bool} \times \mathsf{real}$. The program variables themselves are functions from states to values ($\Sigma \to \alpha$, where α can be any HOL type) and correspond to the projection functions of the state. So when we say $x : \mathsf{num}$ we mean x applied to the state is of type num. In HOL syntax, the three variables in this example are locally bound as follows:

```
let x = FST in let y = FST o SND in let z = SND o SND in
...
```

The programming language originally included while statements, blocks with local variables, and demonic nondeterministic statements, but was later extended with (recursive) procedures with value and reference parameters [11]. All these statements satisfy two basic "healthiness conditions": they are monotonic and conjunctive (c is monotonic if $p \subseteq q$ implies $c\ p \subseteq c\ q$ and c is conjunctive if $c\ (p \wedge q) = (c\ p) \wedge (c\ q)$). We have extended the language (i.e., both the embedding and the parser) with two angelically nondeterministic constructs, which are monotonic but not conjunctive. The weakest precondition for each of the basic statements is defined as follows:

$\mathsf{skip}\ q \ = \ q$	
$\mathsf{abort}\ q \ = \ \mathsf{F}$	
$\mathsf{magic}\ q \ = \ \mathsf{T}$	
$\{p\}\ q \ = \ p \wedge q$	*assertion*
$[p]\ q \ = \ p \Rightarrow q$	*guard*
$(x := e)\ q \ = \ q[x := e]$	*determ. update*
$\{x := x' \| P\}\ q \ = \ (\exists x'.\ P \wedge q[x := x'])$	*angelic update*
$[x := x' \| P]\ q \ = \ (\forall x'.\ P \Rightarrow q[x := x'])$	*demonic update*
$(c_1; c_2)\ q \ = \ c_1\ (c_2\ q)$	*sequence*
$(c_1 \sqcup c_2)\ q \ = \ (c_1\ q) \vee (c_2\ q)$	*angelic choice*
$(c_1 \sqcap c_2)\ q \ = \ (c_1\ q) \wedge (c_2\ q)$	*demonic choice*
$\mathsf{if}\ g\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\ q \ = \ (g \wedge (c_1\ q)) \vee (\neg g \wedge (c_2\ q))$	*conditional*
$[\![\mathsf{var}\ \ y \| b.\ c]\!]\ q \ = \ (\forall y.\ b \Rightarrow c\ q)$	*local block*

(Loops and procedures are covered in Section 3.4 and 3.5.)

The definitions above use the syntax that is recognized by the parser. The following example illustrates how the definitions are realized with the tuple state. The HOL term for the program $\mathsf{skip}; x := x + 1$ in the environment with program variables x and y is as follows:

```
let x = FST in let y = SND in
  skip seq assign (λs.  (x s + 1, y s))
```

In the angelic and demonic updates, $x := x'|P$ describes a state relation which leaves all variables except x unchanged.

3.2 Basic calculations

The calculation of the weakest precondition of programs containing only basic statements with respect to some postcondition is done by simply using the definitions of the statements. For example, calculating the weakest precondition of the above program with respect to postcondition $x = 1$ gives the following theorem:

```
let x = FST in let y = SND in
  (skip seq assign (λs.  (x s + 1, y s))) (λs.  x s = 1)
=
let x = FST in let y = SND in (λs.  x s + 1 = 1)
```

Note the explicit state abstraction in the predicates: the calculated precondition can be simplified to `(λs.  x s = 0)` which in ordinary syntax would simply be written as $x = 0$.

For all the basic statements the calculation gives the weakest precondition of the statement with respect to the postcondition. However, for loops and procedures we will generally not get the weakest precondition, but something stronger (see Section 3.4 and 3.5).

When calculating correctness assertions such as

$$x = 0\{\!|\mathsf{skip}; x := x + 1|\!\}x = 1$$

we do not use Hoare-logic rules. Instead we use the fact that for the correctness assertion to be correct the precondition must be stronger than the weakest precondition of the statement with respect to the postcondition. This will always generate a verification condition. In our example, this verification condition would be

$$\forall x.\ (x = 0) \Rightarrow (x + 1 = 1)$$

(note that the verification condition no longer has a state argument).

Programs and pre- and postconditions can contain free variables, which are implicitly universally quantified in a correctness assertion. An important special case of such variables is that of *specification variables* (also known as auxiliary or logical variables), which are usually used to refer to the initial values of the program variables. They may occur free in preconditions, postconditions, and invariants, but cannot be used in program statements. Free variables that are mentioned by the program can be thought of as *global constants*.

For example, in the following correctness assertion (for a "swapping" assignment) a_0 and b_0 are used to refer to the initial values of a and b:

$$(a = a_0 \wedge b = b_0)\{\!|a, b := b, a|\!\}(a = b_0 \wedge b = a_0)$$

Specification variables are often indicated by special syntactic conventions, such as zero subscription. Another possibility is to make assertions explicitly dependent on the specification variables in addition to the state [14]. In our framework, specification variables require special treatment only when they appear in loops and procedures (see Section 3.4 and 3.5).

3.3 Angelic statements

Angelic and demonic statements can be used together to model interactive systems [1]. Such systems can be thought of as two-player games where one player (the angel) tries to achieve a goal while playing against an opponent (the demon). Proving the correctness of such systems means analyzing the goal achievability for the angel (an example is given in Section 4.1).

The definitions show that angelic statements introduce existential quantifiers or disjunctions in the preconditions. For example, we have

$$(\mathsf{skip} \sqcup x := x + 1)\ (x = 1) \;\; = \;\; (x = 0 \vee x = 1)$$

A consequence of this is that "contradictory" correctness assertions may hold, e.g. both the following:

$$x = 0 \{\!|\mathsf{skip} \sqcup x := x + 1|\!\} x = 1$$
$$x = 1 \{\!|\mathsf{skip} \sqcup x := x + 1|\!\} x = 1$$

and in order to verify a given correctness assertion, a choice between disjuncts must be made (or an existential witness supplied). This typically requires user intervention, and in fact this intervention can be seen as describing a winning strategy for the angel.

3.4 Loops

The loop construct is defined in the usual way, as the least fixpoint (in the predicate transformer lattice) of the unfolding function:

$$\mathsf{do}\ g \rightarrow c\ \mathsf{od} \;\; = \;\; (\mu X.\ \mathsf{if}\ g\ \mathsf{then}\ c; X\ \mathsf{else\ skip\ fi})$$

In order to verify correctness of a loop, we need an invariant and a variant. Furthermore, if an invariant and a variant are given, then the invariant will be a precondition of the loop, although not necessarily the weakest precondition. In existing systems based on precondition calculation, the invariant has generally been used as the calculated precondition of the loop [13]. A well chosen invariant may have to refer to specification variables introduced as initial values for one or more program variables. This is possible, but it can make the annotation of the loop difficult, since the role of the specification variables may not be easy to trace (especially if the loop occurs late in the program text). Another possibility would be to use a "binary invariant", i.e., a relation rather than a predicate (as in VDM [9]) but we use a different solution, which fits well into the classical correctness-based framework.

Our novel solution is based on the following generalization of the variant-invariant theorem for loops, where the invariant I has an extra argument:

$$(\forall s_0\ x.\ (\lambda s.\ I\ s_0\ s \wedge g\ s \wedge (t\ s = x))\ \{|c|\}\ (\lambda s.\ I\ s_0\ s \wedge t\ s < x))$$
$$\Rightarrow$$
$$\forall h.\ (\lambda s.\ I\ (h\ s)\ s \wedge (\forall s'.\ \neg(g\ s') \wedge I\ (h\ s)\ s' \Rightarrow q\ s'))\ \subseteq\ (\mathsf{do}\ g \to c\ \mathsf{od})\ q$$

(The occurrence of $\subseteq$ here indicates that the precondition is not necessarily the weakest one.)

This theorem allows us to introduce specification variables (here s_0, but it can be a tuple of variables) as names for any function (here h) of the initial state, and then the invariant may refer to these specification variables. An important advantage is that this makes loop annotation modular – we can annotate the loop without knowing what context it will be used in. Note also that the requirement that the postcondition must hold upon termination here becomes included in the precondition rather than appearing as a separate verification condition.

For example, the following is a correct loop annotation:

```
var  x, y, z : num.
...
 do /* invariant  (z + x * y = z1 + x1 * y1) ∧ y = y1 variant  x
        init  (x1, y1, z1) */
 0 < x → x, z := x − 1, z + y
 od
```

Here the specification variables are introduced using the init-annotation which indicates that $x1, y1, z1$ stand for the values of the program variables x, y, z at the beginning of the loop (independently of whether specification variables are used in the precondition to stand for their initial values).

Given this loop and the postcondition $z = n$ (where n is a free variable), the tool gives (after some simplification) the precondition $z + x * y = n$.

Having the specification variables stand for initial values of program variables means that the function h in the loop theorem is always instantiated to the identity function. It would be possible to allow more complex specification variables (e.g., standing for the sum of two variables) but that would require a more elaborate annotation scheme.

3.5 Procedures and adaptation

A procedure is a parameterized piece of program that can be used in different contexts. The procedure body is no different from other program statements. Using the procedure, however, requires the adaption of the procedure body to the program context that calls it: actual parameters should be substituted for formal ones. Moreover, a procedure works on a state of its own (the parameter space). The adaption also includes a jump from the calling state to the local procedure state and then back again.

We reuse Laibinis's formalization of procedures [11], which uses a block with local variables corresponding to the value parameters. Procedures modify the part of the global state that is present in the reference parameters list and a pair of inverse functions are responsible for rearranging the state tuple at a procedure call and restoring it after the procedure has exited. The block body works as a skip statement on the part of the global state that is not modified and as the procedure body on the state composed of the value and reference parameters. This is formalized in the definition of the call operator

$$\mathsf{call}\ c\ f\ g\ e$$

where c is the procedure body, f and g are the state-reordering functions and the expression e is the initial value for the value parameters.

Calls to non-recursive procedures can be handled by unfolding the procedures but this requires the procedure body to be analyzed for each procedure call. Our tool also makes this option available but here we concentrate on the modular way of treating procedures, which also gives us a unified way of treating both recursive and non-recursive procedures.

For each procedure our tool proves the procedure body correct with respect to the precondition and postcondition that annotate the procedure definition. Then for each procedure call the pre-proved theorem is adapted by substituting the actual parameters for the formal ones, and lifting the precondition and postcondition to the global state.

As an example, consider the procedure $Swap$ defined as

$$\begin{array}{l}\mathsf{procedure}\ \ Swap(\mathsf{var}\ \ a, b : \mathsf{num})\\ /{*}\ \mathsf{pre}\ \ a = a_0 \land b = b_0\ \mathsf{post}\ \ a = b_0 \land b = a_0\ {*}/\\ \quad a, b := b, a\end{array}$$

For a call with actual parameters x and y the following theorem is generated:

$$\begin{array}{l}\ \ Swap = a, b := b, a\\ \vdash \forall a_0\ b_0.\ (x = a_0 \land y = b_0)\{\!|\ Swap(x, y)\ |\!\}(x = b_0 \land y = a_0)\end{array}$$

Note that the procedure name is a free variable, and the procedure definition is included as an assumption of the theorem.

We need to use this theorem to calculate the precondition of $Swap(x, y)$ with respect to some postcondition over the state of the calling program.

3.5.1 Procedure adaptation

The adaption problem can be described as looking for a rule of the form

$$\frac{p\{\!|c|\!\}q}{?\{\!|c|\!\}r}$$

that would enable us to fill in the missing precondition. The rule is said to be *sharp* if it gives the weakest possible precondition. The fact that specification variables can be present makes this problem nontrivial to solve in a mechanized logic, and the problem has attracted quite a lot of attention recently, e.g., [10, 12]. Our solution uses separate theorems for the monotonic and for the conjunctive case, following [12].

$$\begin{array}{l}\forall c\ p\ q.\ \mathsf{conjunctive}\ \ c \Rightarrow \\ \qquad ((\forall m.\ p(m)\{\!|c|\!\}q(m)) = \\ \qquad (\forall q'.\ (\{\exists m.\ p(m)\}; [x := x' | \forall m.\ p(m) \Rightarrow q(m)])\ q'\ \subseteq\ c\ q'))\end{array}$$

$$\begin{array}{l}\forall c\ p\ q.\ \mathsf{monotonic}\ \ c \Rightarrow \\ \qquad ((\forall m.\ p(m)\{\!|c|\!\}q(m)) = \\ \qquad (\forall q'\ m.\ (\{p(m)\}; [x := x' | q(m)])\ q'\ \subseteq\ c\ q'))\end{array}$$

Here m stands for one or more specification variables that may occur free in the pre- and postconditions p and q. The tool identifies specification variables from the procedure annotation using a simple heuristic: any free variable that occurs on the right-hand side of an equality where the left hand side is a program variable is considered to be a specification variable.

Both theorems are sharp (since they are equivalences). The first theorem is used for conjunctive (non-angelic) statements and the second theorem for statements that cannot be automatically proved to be conjunctive. The left-hand side $(p(m)\{\!|c|\!\}q(m))$ matches the procedure correctness theorem, transported to the calling state (as described above), and q' matches the actual postcondition for the procedure call.

Applying either of the theorems to the *Swap* example gives as expected:

$$(Swap(x, y))(x > z) = (y > z)$$

where z can be a program variable of the calling state or a free variable.

3.6 Recursive procedures

Recursive procedures are often handled by considering the procedure names as a collection of recursively defined constants. In contrast, our tool reuses the recursion operator of the underlying formalization of the refinement calculus. Thus, a recursive procedure is simply an ordinary procedure of the form $P = (\mu X.S[X])$ and the procedure name is a free variable. Separating procedures and recursion simplifies the treatment of recursive procedures, but since the recursive occurrence of the recursion variable is as an argument to call, the separation is not complete.

Exactly as for loops, we have proved a theorem for correctness of recursion using an invariant and a variant, and specification variables:

$$
\begin{array}{l}
(\forall s_0\ x.\ (\lambda s.\ p\ s \wedge (h\ s = s_0) \wedge (t\ s = x)) \\
\qquad \{\!|f(\{\lambda s.\ p\ s \wedge t\ s = x\}; [\lambda s\ s'.\ Q\ (h\ s)\ s'])|\!\}Q\ s_0) \\
\Rightarrow \\
\forall s_0.\ (\lambda s.\ p\ s \wedge (h\ s = s_0))\{\!|\mu f|\!\}Q\ s_0
\end{array}
$$

Here the invariant and the variant are expressed in terms of the parameters at the point of the recursive calls and specification variables that are local to the procedure. Instead of having an induction assumption about correctness of the recursive call, the antecedent in this rule has an explicit nondeterministic statement at the recursion point (the argument to the function f) which states that the variant is decreased and the postcondition established, taking into account that specification variables now refer to the values of the program variables at the recursive call.

Exactly as in the loop theorem, the specification variables (s_0 in the theorem) can stand for any function (h) of the initial state. In practice, the heuristic used to identify specification variables restricts them to stand for initial values of program variables.

For an example of a recursive procedure, see Section 4.2 below (procedure *Add*).

3.7 The procedure database

We store the proved facts about individual procedures in a database. The database entries are indexed by the procedure names and types. Each entry contains a procedure definition and a theorem expressing the correctness of the procedure body with respect to the precondition and postcondition that annotate the procedure definition.

The entries also contain theorems about the healthiness of the procedures. Theorems about the monotonicity of individual procedures are added because the shallow embedding of the language makes it impossible to prove the monotonicity of our program statements once and for all. The actual HOL type of our program statements includes all predicate transformers, the monotonic and non-monotonic ones. Conjunctivity on the other hand depends on whether the procedures have only demonic nondeterminism or not, so it cannot be proved for all statements. Note that monotonicity and conjunctivity of procedures are needed when using the adaption theorems.

Example of a database entry:

$$((Swap : (\mathsf{num} \times \mathsf{num} \to \mathsf{bool}) \to (\mathsf{num} \times \mathsf{num} \to \mathsf{bool})),$$
$$(Swap = a, b := b, a),$$
$$(Swap = a, b := b, a \vdash \forall a_0\, b_0.\ (a = a_0 \wedge b = b_0)\{\!|\ Swap\ |\!\}(a = b_0 \wedge b = a_0)$$
$$(Swap = a, b := b, a \vdash \mathsf{monotonic}\ \ Swap),$$
$$(Swap = a, b := b, a \vdash \mathsf{conjunctive}\ \ Swap))$$

The main reason for having a procedure database is to avoid repeatedly calculating correctness and proving healthiness for the same procedure. Also, declaring the same procedure in more than one program becomes unnecessary since all the programs can use the theorems contained in the database. Another benefit is that the program terms become smaller and the calculation faster if the (typed) procedure names are used instead of the procedure body terms. The verification conditions also become smaller (though there will be more of them) and they are more intelligible when we can tell what part of the program they come from.

3.8 Simplification

Each generated verification condition, or precondition is simplified using the basic HOL simplifier (with the built-in `hol_ss` simpset). For all but trivial examples, this simplification is not sufficient to discharge all the proof obligations. The difficulty in proving verification conditions stems from the intricate data types needed in realistic programs, and the myriad of program domains. Ideally, different specialized simplifiers should be used for different programs.

However, there is a layer of specialized simplification that can be performed, by exploiting typical patterns that repeatedly appear in different verification conditions. For example, one of the typical conditions has to do with the loop variant decreasing for each iteration and could take this form:

$$\cdots 0 < x \cdots \Rightarrow \cdots \wedge (x - 1 < x) \wedge \cdots$$

Similarly, demonic and angelic updates give rise to quantified conditions that can be simplified using certain sequences of rewrites and quantifier manipulations. We specialize the more general HOL simplifier with some of these patterns in mind. This simplification is performed in a separate step in order to avoid infinite looping during the main precondition/correctness calculation.

4 Examples

The examples in this section demonstrate some of the features of our tool. The first example illustrates game-like programs with angelic and demonic nondeterminism, the second example shows procedures at work, while the third example contains multiple loops and program variables with array types.

4.1 Angelic and demonic nondeterminism

In this example we model the game of Nim and prove a correctness assertion about it. In Nim two players take turns in removing one or two matches from a pile. The player to remove the last match loses the game. We side with the first player, hence the first player is interpreted as the angel and the other player as the demon. x is the number of matches. We prove that the angel can win the game, provided that initially $x \;\mathsf{MOD}\;\; 3 \neq 1$, by making sure that he removes the matches so that $x \;\mathsf{MOD}\;\; 3 = 1$ after his turn.

$$
\begin{array}{l}
\mathsf{var}\;\; x : \mathsf{num}. \\
\neg(x \;\mathsf{MOD}\;\; 3 = 1)\{| \\
\;\mathsf{do}\; /\!*\; \mathsf{invariant}\;\; \neg(x \;\mathsf{MOD}\;\; 3 = 1)\; \mathsf{variant}\;\; x \;*\!/ \\
\;\mathsf{T} \rightarrow \\
\quad\;\; [0 < x]; \{x := x' | (x' \;\mathsf{MOD}\;\; 3 = 1) \wedge (x' = x - 1 \vee x' = x - 2)\}; \\
\quad\;\; \{0 < x\}; [x := x' | x' = x - 1 \vee x' := x - 2] \\
\;\mathsf{od} \\
|\}\mathsf{T}
\end{array}
$$

The guard $[0 < x]$ is interpreted as: check if there are matches before the angel takes the turn and if not terminate miraculously with the angel winning the game. Similarly, the assertion $\{0 < x\}$ is interpreted as: the demon wins if there are no more matches when he takes the turn.

The correctness proof generates one verification condition which after basic simplification looks as follows:

$$
\begin{array}{l}
\forall x. \\
\;\; \neg(x \;\mathsf{MOD}\;\; 3 = 1) \Rightarrow \\
\quad\; 0 < x \Rightarrow \\
\quad\;\; (\exists x'.\, (x' \;\mathsf{MOD}\;\; 3 = 1) \wedge (x' = x - 1 \vee x' = x - 2) \wedge 0 < x' \wedge \\
\qquad\; (\forall x''.\, (x'' = x' - 1 \vee x'' = x' - 2) \Rightarrow \neg(x'' \;\mathsf{MOD}\;\; 3 = 1) \wedge x'' < x))
\end{array}
$$

This verification condition contains the core of the correctness proof obligation. It should be read as: provided that the precondition holds and there are matches in the pile there exists a winning strategy for the angel regardless of what the demon chooses. The witness for the existential quantifier gives, in fact, the winning strategy for the angel. If x $\mathsf{MOD}\ \ 3 = 0$ the angel should remove two matches, and if x $\mathsf{MOD}\ \ 3 = 2$ the angel should remove one match. Our specialized simplifier removes quantifiers and the condition that comes from the variant $(x'' < x)$ but the complete proof involves more specialized reasoning about the MOD operator and has to be separately carried out in HOL.

4.2 Nested and recursive procedures

The following program multiplies two natural numbers by calling a procedure *Mult*. *Mult* calculates the product using a loop which calls the procedure *Add*, which is recursive and calls simple procedures for incrementing and decrementing variables.

$$
\begin{array}{l}
\mathsf{procedure}\ \ Inc(\mathsf{var}\ \ a : \mathsf{num}) \\
/*\ \mathsf{pre}\ \ a = a_0\ \mathsf{post}\ \ a = a_0 + 1\ */ \\
\quad a := a + 1 \\
\mathsf{procedure}\ \ Dec(\mathsf{var}\ \ a : \mathsf{num}) \\
/*\ \mathsf{pre}\ \ 0 < a \wedge a = a_0\ \mathsf{post}\ \ a = a_0 - 1\ */ \\
\quad a := a - 1 \\
\mathsf{procedure}\ \ Add(\mathsf{val}\ \ a : \mathsf{num}\ \mathsf{var}\ \ b : \mathsf{num}) \\
/*\ \mathsf{pre}\ \ a = a_0 \wedge b = b_0\ \mathsf{post}\ \ b = b_0 + a_0 \\
\quad \mathsf{invariant}\ \ b = b_0 + (a_0 - a)\ \mathsf{variant}\ \ a\ */ \\
\quad \mathsf{if}\ 0 < a\ \mathsf{then}\ Inc(b); Dec(a); Add(a, b)\ \mathsf{else\ skip\ fi} \\
\mathsf{procedure}\ \ Mult(\mathsf{val}\ \ a : \mathsf{num}\ \mathsf{var}\ \ b : \mathsf{num}) \\
/*\ \mathsf{pre}\ \ a = a_0 \wedge b = b_0\ \mathsf{post}\ \ b = b_0 * a_0\ */ \\
\quad [\![\mathsf{var}\ \ z : \mathsf{num} | z = b. \\
\quad b := 0; \\
\quad \mathsf{do}\ /*\ \mathsf{invariant}\ \ z = Z \wedge (b + a * z = B + A * z)\ \mathsf{variant}\ \ a \\
\qquad\quad \mathsf{init}\ \ (Z, A, B)\ */ \\
\quad 0 < a \rightarrow Add(z, b); Dec(a)\ \mathsf{od} \\
\quad]\!] \\
\mathsf{var}\ \ x, y : \mathsf{num}.\ Mult(x, y)\ (y = z)
\end{array}
$$

Add is a recursive procedure so an invariant and variant for the recursion are needed (note that the invariant refers to the specification variables introduced in the precondition).

The calculated precondition is $x*y = z$ and the two verification conditions that were not proved by the basic simplifier of the tool are:

$$vc_1: \quad \forall a_0, b_0, a, b.\ \neg(0 < a) \wedge (b + a * b_0 = a_0 * b_0) \Rightarrow (b = b_0 * a_0)$$

$$vc_2: \quad \forall A, B, Z, a, b.\ (b{+}a{*}Z = B{+}A{*}Z) \wedge 0 < a \Rightarrow (Z{+}b{+}(a{-}1){*}Z = B{+}A{*}Z)$$

and they are both discharged by our more specialized simplifier.

4.3 Arrays and multiple loops

The following program finds the least number that does not occur in an array $a[0..n-1]$ of natural numbers, by first noting in a separate boolean array b which numbers occur in a. Arrays of type α are modeled as functions $\mathsf{num} \to \alpha$. Note that the loop annotations are sufficient for the tool to verify correctness:

$\mathsf{var}\ \ i : \mathsf{num}; a : \mathsf{num} \to \mathsf{num}; b : \mathsf{num} \to \mathsf{bool}.$
$\mathsf{T}\{\!|$
$i := 0$
do /* $\mathsf{invariant}\ \ i \le n \wedge (\forall k.\ k < i \Rightarrow \neg b[k])\ \mathsf{variant}\ \ n - i$ */
$i < n \to b := (\lambda k.\ (k = i) \to \mathsf{F} \mid b[k]); i := i + 1$
od
$i := 0;$
do /* $\mathsf{invariant}\ \ i \le n \wedge (\forall k.\ k < i \Rightarrow (b[k] = (\exists j.\ j < i \wedge a[j] = k)))$
$\quad\quad \mathsf{variant}\ \ n - i$ */
$i < n \to b := (\lambda k.\ (k = a[i]) \to \mathsf{T} \mid b[k]); i := i + 1$
od
$i := 0;$
do /* $\mathsf{invariant}\ \ i \le n \wedge (\forall k.\ k < i \Rightarrow b[k])\ \mathsf{variant}\ \ n - i$ */
$i < n \wedge b[i] \to i := i + 1$
od
$|\!\}\, i \le n \wedge (\forall j.\ j < n \Rightarrow \neg(a[j] = i)) \wedge (\forall k.\ k < i \Rightarrow (\exists j.\ j < n \wedge a[j] = k))$

The proof generates four verification conditions and fairly elaborate interactive proof is required to discharge them.

5 Conclusion

We trust that a tool for automatically extracting the conditions for which a program is correct with respect to a precondition and a postcondition can mitigate the laborious task of proving program correctness. Although the idea is not new, there are hardly any tools that can handle both realistic languages and realistic program variable types. Specification variables are also handled in varying degrees. The most interesting tool in both respects we found to be Filliatre's [3]. Homeier's [8] is interesting in that it is the only one to also handle mutually recursive procedures.

Our HOL-based tool automatically generates the conditions for which a program is correct with respect to a pre- and postcondition by using weakest preconditions to prove a correctness theorem with the verification conditions as assumptions. Our tool fares well with respect to the complexity of the language and that of the program variable types. Program variables can be of arbitrary types and the language, unlike the other mentioned ones, has angelic and demonic nondeterministic statements, which are suitable for modeling interactive systems. The two new (semantically derived) rules for calculating loop preconditions and correctness of recursive statements take into consideration the presence of specification variables in the invariants.

The modular way we treat programs with procedures makes it easier to verify large programs. The fact that also loops can be handled modularly makes the tool more usable in the sense that we could extract the more difficult parts of the programs and prove them separately. A useful extension would be to permit (but not require) intermediate "helpful" assertions as annotations in sequences. Such an approach might also make the verification conditions more understandable.

The soundness of the verification condition generators is an important issue. Nipkow [13] and Homeier [8] prove their verification condition generators sound. Our tool is an unverified HOL/ML-program, but since the result of each precondition/correctness calculation is a theorem, we get soundness for free.

References

[1] R. J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.

[2] M. J. Butler, J. Grundy, T. Långbacka, R. Rukšenas, and J. von Wright. The refinement calculator: Proof support for program refinement. In *Proc. FMP'97 - Formal Methods Pacific*, Discrete Mathematics and Theoretical Computer Science, Wellington, New Zealand, July 1997. Springer-Verlag.

[3] J. C. Filliatre. Proof of imperative programs in Type Theory. In *International Workshop, TYPES'98*, Kloster Irsee, Germany. LNCS 1657, Springer-Verlag, March 1998.

[4] J. C. Filliatre and N. Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[5] N. Francez. *Program Verification.* Addison-Wesley, 1992.

[6] M. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistel and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.

[7] M. J. C. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge University Press, New York, 1993.

[8] P. V. Homeier. Trustworthy tools for trustworthy programs: A mechanically verified verification condition generator for the total correctness of procedures. Ph.D. Dissertation, UCLA Computer Science Department, 1995.

[9] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.

[10] T. Kleymann. Hoare Logic and VDM: Machine-checked soundness and completeness proofs. (Phd Thesis), ECS-LFCS-98-392, FCS, 1998.

[11] L. Laibinis. Mechanized formal reasoning about modular programs. Turku Centre for Computer Science Dissertations, 24, Finland, April, 2000.

[12] L. Laibinis and J. von Wright. What's in a specification. In *Proc. IRW/FMP'98 - International Refinement Workshop and Formal Methods Pacific 1998*, Discrete Mathematics and Theoretical Computer Science, Canberra, Australia, September 1998. Springer-Verlag.

[13] T. Nipkow. Winskel is (almost) Right: Towards a Mechanized Semantics Textbook. *Formal Aspects of Computing 10*, pp. 171-186, 1998.

[14] D. von Oheimb. Hoare Logic for Mutual Recursion and Local Variables. In: *Foundations of Software Technology and Theoretical Computer Science (FST and TCS)*, Springer LNCS, 1999.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi

University of Turku
- **Department of Information Technology**
- **Department of Mathematical Sciences**

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

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**