Mechanizing Program Verification in HOL

Sten Agerholm
Computer Science Department
Aarhus University
Denmark

Abstract

Proofs of program correctness are usually large and complex. This advocates mechanical assistance for managing the complexity and details of proofs. This paper presents a program verifier for imperative programs based on the HOL system. We describe a formalization of the weakest precondition semantics of a small programming language, a verification condition generator for total correctness specifications, and a number of simplification tools for proving subparts of verification conditions, automatically. Examples are considered in order to evaluate the usability of the program verifier.

1 Introduction

The field of program correctness really got started in the 1960s due to papers by Floyd [9] and Hoare [13]. In the paper by Hoare (inspired by Floyd, however) a small programming language was defined in terms of a logical system of axioms and inference rules, called (Floyd-) Hoare logic, for proving program correctness. In 1976 E.W. Dijkstra presented a wide collection of programming methods and tools in his book [7]. Dijkstra showed how a mixture of common sense and formal/informal reasoning could lead to correct (more reliable) programs. In the book, a small programming language was defined by giving a weakest precondition for each construct w.r.t. a postcondition.

The advantages of Dijkstra's work are that he takes a more mathematical approach to program semantics, and provide methods and tools for developing correct programs. However, often Dijkstra-style correctness proofs are rather informal (see [7, 12]), and therefore do not really suit mechanical verification. On the other hand, Hoare's logic of axioms and inference rules suits mechanical verification since axioms and rules can be applied systematically in strict and formal proofs about program specifications. The underlying ideas of many program verifiers are therefore based on (Floyd-) Hoare logic, with postulated axioms and rules used backwards in proofs of specifications.

The use of the HOL system [10] to verify imperative programs has been investigated in Gordon [11] and Back and von Wright [5]. Gordon gives a relational semantics of a small programming language and derives a Hoare logic, on top of which a verification condition (vc) generator is implemented. The problem of proving vc's is not treated. Back and von Wright describe a formalization of the refinement calculus which is based on Dijkstra's weakest preconditions and transformational approaches to program construction and verification [2, 3]. However, they do not provide the tools needed for verifying actual programs.

There is one main problem with these works: all program variables of a program must have the same type. It was therefore our goal to obtain a formalization of programming language semantics in HOL that allows programs on any combination of application domain theories. Another important goal was to provide a programming language general enough to support the stepwise refinement of programs from specifications in the refinement calculus. In fact, the language considered here is (essentially) the same as the language in the paper [5], i.e. it is a generalization of Dijkstra's language of guarded commands, extended with nondeterministic assignments and blocks. However, the formalization of it in the HOL system is quite different (see the acknowledgements). Finally, we aimed at providing a practical program verifier based on the HOL system.

We try to combine the advantages of both the mathematical approach of Dijkstra and the axiomatic approach of Hoare. Like Dijkstra, we define the weakest precondition semantics of our programming language constructs, called commands. We then prove various properties of the semantics (Dijkstra's healthiness conditions). However, the development of a presumably correct program is probably done most conveniently outside the HOL system (by hand, or what-

0-8186-2460-4/92 $03.00 © 1992 IEEE

ever). The program specification can then be formally verified using the HOL system. A verification condition generator was implemented for that purpose which can be used to reduce total correctness specifications to a number of pure boolean logic terms, called verification conditions. Following ideas in the paper [11], the implementation is based on Hoare-like axioms and rules which have been formally derived from the semantics of commands in HOL (not postulated), and thus it is guaranteed by the HOL system that theorems proved about programs are logical consequences of the underlying programming language semantics.

Unfortunately, proving the vc's is difficult, both due to the reasoning they require and due to the fact they are a large collection of complex terms. In order to reduce the workload of the 'human' verifier, it is therefore desirable to have tools to, if not prove verification conditions, then at least simplify them such that only non-trivial parts are left to be proved, interactively. The simplification tools are intended to prove subpart of the vc's, automatically.

1.1 Organization of the Paper

The rest of the paper is organized as follows. Section 2 describes a number of HOL theories needed for the formalization of weakest precondition semantics. The section describes the chosen representation of states, and theories for predicates, predicate transformers and extreme solutions (fixpoints). In section 3 the programming language considered is formalized in HOL. Among other things, the section describes conversions for calculating boolean term versions of weakest preconditions and an interface for HOL extended with commands. In section 4 the derived Hoare-like rules are presented. For the iteration rule a theory of well-founded sets was formalized, and is also described there. Section 5 describes the implementation of the verification condition generator which is based on tactics built on top of the Hoare-like rules. The section introduces an alternative to annotating specifications, called labelling, where labels are associated with points in the program text. In section 6 the simplification tools are described. Among other things, resolutions tactics are described which use hints to restrict acceptable resolvents, and calculate new hints for subgoals. In section 7 the program verifier is used to verify three small examples in order to evaluate its usability. Finally, section 8 provides some conclusions.

A more thorough treatment of the work presented here can be found in [1].

1.2 Notation

We use the usual syntax of HOL terms except that 'λ' is used for '\', '$\forall$' for '!' and '$\exists$' for '?'.

2 Preliminary Theories

In this section we describe a number of theories that were needed for formalizing the weakest precondition semantics of the programming language constructs.

2.1 Predicates

A state is an assignment of values to program variables. We view a state as a tuple of values in a Cartesian product space, also called a state space. Each program variable is associated with a certain component of the state space.

A predicate is a function from a state space to the boolean truth values. Thus, it can be regarded as a condition which is either true or false in a state, or as a set of states. If `p` is a function of type `":ty1#...#tyn -> bool"` we can say that `p` is a predicate on the state space `":ty1#...#tyn"`, or that this is the state space of `p`. Given variables `x1`, ..., `xn` which are associated with components of this state space explicitly or implicitly, we say that `p` is a predicate on the program variables `x1`, ..., `xn`, or that these are the program variables of `p`. Variables in a predicate which are not among the program variables are called logical variables. For instance, the predicate

```
"λ(x,y). x<y /\ z"
```

has two program variables `x` and `y` and a logical variable `z`. The program variables are associated with components of the state space explicitly, due to the λ-abstraction.

Often we do not want to consider one particular state space but *any* state space. If we use a polymorphic type `":*s"` as the state space we are able to do so. Below, this is the state space considered unless another space is obvious from the context. The type `":pred"` abbreviates the type of predicates on the state space `":*s"` (not a valid HOL abbreviation). The use of polymorphism allows all theories used in formalizing program semantics to be independent of program variables and their types. Thus, the theories can be developed once and for all, and later used for any programs and program variables.

Various operators on predicates have been defined. They are represented by higher order HOL constants,

lifting the corresponding boolean operators by pointwise extension. For instance, the nullary operator for predicate truth is defined by the defining theorem

```
⊢ true  = (λv. T)
```

and predicate conjunction is defined by

```
⊢ ∀p q. p and q = (λv. p v /\ q v).
```

Predicate operators for falsity (**false**), negation (**neg**), disjunction (**or**), implication (**imp**) and equality (**equals**) have been defined, similarly. Since the operators are defined in a systematic way, it was possible to automate proofs of theorems involving these operators by lifting boolean theorems and various conversions and inference rules.

Universal and existential quantification over logical variables are also provided. Universal quantification is introduced by the theorem

```
⊢ ∀f. forall f = (λv. ∀x. f x v),
```

where the type of `f` is `":* -> pred"`. Existential quantification is defined in a similar way.

Two operators are introduced for changing the state space of a predicate. The extension of a predicate is defined by

```
⊢ ∀R p. ext R p = (λu. p(R u)).
```

Thus, given a certain 'selector' function `R` of type `":*es -> *s"` and a predicate on the state space `":*s"`, `ext` returns a predicate on the extended state space `":*es"`. The restriction of a predicate is defined by

```
⊢ ∀A p. restr A p = (λv. ∀w. p(A w v))
```

where the type of `A` is `":*e -> *s -> *es"`. As it appears, we restrict a predicate by quantifying, universally, over the state space components we want to get rid of.

Quite often we need to express the property that a predicate holds for all states of the state space. In order to express things like this neatly, we introduce the *everywhere* operator **ew**. Applying it to a predicate `p`, we obtain a boolean term `"ew(p)"` which expresses that 'predicate `p` holds everywhere'. Not surprisingly, its definition looks as follows

```
⊢ ∀p. ew(p) = (∀v. p v)
```

where the universal quantification over the state space appears, explicitly.

2.2 Predicate Transformers

A predicate transformer is a a function from predicates to predicates on the same state space. The notions of predicate on a state space and on program variables are generalized to predicate transformers. Below, we use `":ptrans"` to abbreviate the type `":pred -> pred"` of predicate transformers on the state space `":*s"` (not a valid HOL abbreviation).

A theory of properties of predicate transformers was formalized in HOL. For instance, we defined the monotonicity property by the defining theorem

```
⊢ ∀f.
    monotonic f =
    (∀p q.
      ew(p imp q) ==> ew((f p) imp (f q)))
```

which says a monotonic predicate transformer preserves predicate implication, and the conjunctivity property by

```
⊢ ∀f.
    conjunctive f =
    (∀p q. f(p and q) = (f p) and (f q)).
```

which says that a predicate transformer is conjunctive if it distributes over conjunctions of predicates.

A number of theorems have been proved about the properties of predicate transformers. For instance, the theorem

```
⊢ ∀f g.
    conjunctive f /\ conjunctive g ==>
    conjunctive(λp. (f p) and (g p))
```

which states that combining conjunctive predicate transformers by conjunction preserves conjunctivity. Such facts are used to prove that properties such as Dijkstra's healthiness conditions hold for the semantics of commands.

2.3 Extreme Solutions

In order to describe the semantics of iteration we formalized a theory of equations in predicates and their extreme solutions (based on [8]). An equation can have any number of solutions but we are concerned with the strongest and weakest solutions, only, also called the extreme solutions. Strongest and weakest solutions of certain equations correspond to least and greatest fixpoints. In fact, a theory of least fixpoints was derived as a special case of the theory of extreme solutions.

We are interested in solutions `q` of equations like

```
"f q = true"
```

where `f` is a predicate transformer. In the future we will confuse `f` with the equation above. That is, if we say that a predicate `q` is a solution of an equation `f` we really mean that `q` solves the equation `"f q = true"`.

An equivalent way of writing the equation is as the term `"ew(f q)"`. This is easier to work with in proofs and therefore used in the following definition

```
⊢ ∀q f. q iso f = ew(f q)
```

which introduces the constant `iso` to capture the statement that a predicate is a solution of an equation.

Not all equations `f` have solutions. However, if `f` is an equation

```
"λp. (g p) imp p"
```

or an equation

```
"λp. (g p) equals p"
```

where `g` is monotonic, there are extreme solutions, given by the Knaster-Tarski theorem (proved in HOL) — furthermore this theorem gives that extreme solutions of the two kinds of equations are the same. Precisely, the strongest solutions are calculated by

```
⊢ ∀f. sso f = (λv. ∀q. q iso f ==> q v),
```

which, for an arbitrary equation `f`, defines the term `"sso f"` to be the intersection of all solutions of `f`. We have proved that the term `"sso f"` indeed does calculate a strongest solution if one exists, and that it determines the strongest solutions uniquely. Weakest solutions have also been defined, and were proved to be the duals of strongest solutions.

Among other main results about extreme solutions we have proved the theorem

```
⊢ ∀f.
    conjunctive2 f ==>
    conjunctive
    (λp. sso(λp'. (f p p') equals p'))
```

where the quantified variable `f` has the type `":pred -> pred -> pred"` and the term `"conjunctive2 f"` expresses `f` is conjunctive in both predicate parameters. The theorem can be interpreted as stating that strongest solutions viewed as predicate transformers in a free variable preserve conjunctivity of (certain) equations involving equality. The theorem is important because it is used to prove that the iterative command preserves conjunctivity of its subcommand (the paper [5] failed to obtain this result). Properties of weakest solutions were used for the proof of this fact.

3 Formalizing Program Semantics

The syntax and semantics of a small programming language has been formalized in the HOL system. The language is essentially Dijkstra's language of guarded commands [7, 12], extended with a nondeterministic assignment command and a block command [2, 3, 4]. These introduce unbounded nondeterminism into the language and therefore Dijkstra's original semantics of iteration cannot be used (see [8]).

3.1 A language of Commands

The syntax of the language of commands we consider can be described recursively as follows:

```
C  ::=  {P}
     |  w:= w'.P
     |  [var w; C]
     |  C1;C2
     |  if B1 -> C1 [] B2 -> C2 fi
     |  do B -> C od
```

where `P`, `B`, `B1` and `B2` are 'predicates' (boolean logic terms written in HOL syntax), `w` and `w'` are tuples of variables, and `C`, `C1` and `C2` are commands. This is basically the same language as provided in [5] though our formalization of it is very different.

Assuming a tuple of variables `v` and a command `C`, we introduce the syntax `v.<C>` to make explicit, or to *declare*, which variables of `C` are the program variables. It can be read as "`C` is a command on the program variables `v`". The variables `v` should include at least all variables used in blocks (the third construct) and on the left-hand side of assignments (the second construct). The variables in `C` which are not among the program variables `v` are called logical variables.

The last three commands are the usual commands for sequential composition, conditional composition and iterative composition. The first command is called the assert command and it is used to assert the truth of a predicate `P`. Thus, it aborts if `P` does not hold, and terminates without affecting the program variables otherwise. The second command is called the nondeterministic assignment command. It assigns the values of local (logical) variables `w'` to program variables `w` (pairing the tuples componentwise) such that a predicate `P` is satisfied. The nondeterministic assignment command is an important specification command which can be used to express arbitrary input-output specifications. Finally, the third command is

called the block command and it is used to introduce new local program variables for the subcommand.

3.2 Weakest Precondition Semantics

We give the weakest precondition semantics of each programming language construct. The weakest precondition of a command `C` with respect to a predicate `q` is defined to be the set of all initial states such that execution of `C` begun in any one of them is guaranteed to terminate in a state satisfying `q`. Thus, the semantics associates a predicate transformer with each command which given a predicate, interpreted as a postcondition, gives an expression for the weakest precondition. The semantics is therefore also called the weakest precondition predicate transformer semantics.

The commands introduced above are syntactical entities, and not valid HOL terms. We formalize the commands in HOL by representing each command construct as a higher order HOL constant, and by defining each constant to be equal to a weakest precondition predicate transformer, in keeping with the intended semantics. Notions as command on a state space and command on program variables are obtained by viewing commands as predicate transformers.

The syntax and semantics of the command language are introduced into the HOL system by the following definitions:

```
⊢ ∀p q. assert p q = p and q
⊢ ∀M P q.
   nonda M P q =
   (exists(λw. P w)) and
   (forall(λw. (P w) imp (λv. q(M w v))))
⊢ ∀A R c q.
   block A R c q = restr A(c(ext R q))
⊢ ∀c1 c2 q. (c1 seq c2)q = c1(c2 q)
⊢ ∀b1 c1 b2 c2 q.
   if b1 c1 b2 c2 q =
   (b1 or b2) and
   (((not b1) or (c1 q)) and
    ((not b2) or (c2 q)))
⊢ ∀b c q.
   do b c q =
   sso
   (λp. ((b or q) and
         ((not b) or (c p))) equals p)
```

In order to understand these constant definitions easierly we can use them to provide a semantics of the syntactical commands described above. The denotation of the syntax of commands can be described recursively as follows (these recursive transformations could be carried out by a parser and pretty-printer in order to provide a more user-friendly interface):

```
⟦v.<{P}>⟧ = "assert(λv. P)"
⟦v.<w:= w'.P>⟧ =
   "nonda(λw' v. v[w'/w])(λw' v. P)"
⟦v.<[var w; C]>⟧ =
   "block(λw v. v•w)(λv•w. v)⟦v•w.<C>⟧"
⟦v.<C1;C2>⟧ = "⟦v.<C1>⟧ seq ⟦v.<C2>⟧"
⟦v.<if B1 -> C1 [] B2 -> C2 fi>⟧ =
   "if(λv. B1)⟦v.<C1>⟧(λv. B2)⟦v.<C2>⟧"
⟦v.<do B -> C od>⟧ = "do(λv. B)⟦v.<C>⟧"
```

where syntax is as above. We have extended the meta syntax for substitution to substitution on tuples so that this is done componentwise. For instance, the term `"v[w'/w]"` denotes the term obtained from `v` by substituting the i'th component of `w'` for the i'th component of `w`, for all i; of course, the tuples `w` and `w'` must have the same type and length. We also need to explain that the notation `v•w` stands for the tuple obtained by combining the tuples `v` and `w`.

Now, let us return to the semantics of commands. The sequential composition and conditional composition commands have their usual meanings, and their weakest precondition predicate transformer semantics is defined as usual (see e.g. [7, 12]). Iteration also has its usual meaning, and is defined to be the strongest solution of a certain equation, as in [8] (equivalent to a least fixpoint definition). The assert command, written `assert p`, asserts the truth of the predicate `p`, and its semantics is straightforward (see e.g. [4]).

The block command is used to introduce one or more new (uninitialized) variables which are local to a command. Thus, it will temporarily extend the state space to include the new variables, execute the associated command on the extended space and then restrict the state space back to the original one, ignoring the added dimensions. Its semantics is therefore defined using the operators for changing the state space (cf. section 2.1). The HOL type of the new constant introduced for blocks is

```
"block:(*e -> *s -> *es) -> (*es -> *s) ->
       eptrans -> ptrans",
```

where `":*e"` denotes the intended extension of the state space `":*s"`, `":*es"` denotes the extended state and `eptrans` is used for a predicate transformer on `":*es"`. Note, this is just a free interpretation of the types which in fact are completely unrelated. In practice, it is therefore required that the first two parameters of the block constant in the term `"block A R c"` where `c` is some appropriate command, are related in the following way

```
⊢ ∀v w. R(A w v) = v,
```

in order for the predicate transformer "`block A R c`" to define the desired block command. This is the case in the use of `block` to define the meaning of the syntactical construct `v.<[var w; C]>`.

Finally, the nondeterministic assignment command is used to assign the values of new local variables to a number of program variables such that the associated predicate is satisfied (the second parameter, called `P` in the definition above, is a predicate when it has been applied to the local variables). If this is not possible the command aborts (due to `exists`). The syntax of the nondeterministic assignment command in HOL is also somewhat complicated. The first parameter can be interpreted to represent the left-hand side of a nondeterministic assignment command (in the original syntax), and the second parameter to represent the right-hand side. Let us consider an example. Assume three program variables `x`, `y` and `z` and assume we want to assign `y` some value greater than both `x` and `z`. This command can be written as `y:= y'.(x<y' /\ z<y')` in the usual syntax (ignoring the declaration), and the corresponding command is written in HOL as follows

```
"nonda(λy' (x,y,z). x,y',z)
      (λy' (x,y,z). x<y' /\ z<y')".
```

Thus, the local variable of the command replaces the variable which must be assigned a value in the body of the first argument of `nonda`. In this way the value of the local variable is transferred to the program variable associated with the second component of the state space, which is `y`. Multiple nondeterministic assignments are also possible.

From the example it appears that the nondeterministic assignment command introduces unbounded nondeterminism into the command language. If the subcommand of a block command uses the values of uninitialized local variables, unbounded nondeterminism is also introduced, provided of course the set of possible values of some local variable is infinite.

3.3 Derived Commands

The derived commands that we consider are defined as follows

```
⊢ skip = assert true
⊢ abort = assert false
⊢ ∀E.
    assign E =
    nonda(λw v. w)(λw v. w = E v)
```

from which the usual definitions of skip and abort have been derived as theorems

```
⊢ ∀q. skip q = q
⊢ ∀q. abort q = false
```

The definition of the (multiple) assignment command has been simplified to

```
⊢ ∀E q. assign E q = (λv. q(E v)).
```

For example, a multiple assignment which we would typically write as `x,z:= x+y,T`, could be written in HOL as follows

```
"assign(λ(x,y,z). x+y,y,T)"
```

assuming `x`, `y` and `z` are the program variables. The syntactical entities introduced for commands above can be extended with new syntax for these commands in a straightforward way.

Note that the textual substitution usually used in defining the predicate transformer for assignment (see for instance [7, 12]) is replaced by functional application in the λ-calculus. The actual substitution is handled by β-conversion of higher order logic. Also note, the difference between nondeterministic and usual assignment appears quite clearly, above. The usual assignment corresponds to a function mapping old states to new states, whereas the nondeterministic assignment command corresponds to a relation specifying how new states are related to old states.

3.4 Properties of Commands

In order to motivate that the semantics of commands defines reasonable constructions, four properties of the semantics should hold. These are called Dijkstra's healthiness conditions [7] (see also [12]). Actually, Dijkstra's healthiness conditions included a fifth property, called or-continuity. However, this property is often left out since it does not hold in the presence of unbounded nondeterminism.

The four conditions require that the commands (viewed as predicate transformers) should be monotonic, conjunctive, weakly disjunctive and strict. A command is monotonic if it preserves predicate implication, and it is conjunctive if it distributes over predicate conjunction (cf. section 2.2). A command `c` is said to be strict if "`c false = false`". This property is also called the law of excluded miracle since it would indeed be a miracle if a command could terminate in a state satisfying `false`. A command `c` is weakly disjunctive if for any predicates `q` and `r`,

it is true that "ew(((c q) or (c r)) imp (c(q or r)))".

Dijkstra's healthiness conditions have been shown to hold for the semantics of commands. Basic commands were proved to satisfy the properties, and recursively defined commands were proved to preserve the properties of subcommands. For instance, the theorem

```
⊢ ∀E. conjunctive(assign E)
```

states the assignment command is conjunctive (derived as a special case of the conjunctivity of nondeterministic assignment), and the theorem

```
⊢ ∀c.
    conjunctive c ==>
    (∀b. conjunctive(do b c))
```

states the do-command preserves conjunctivity.

A meta theorem (conversion) was obtained for each property by instantiating one general function with the corresponding collection of theorems. For instance, we have the functions CMDS_MONO and CMDS_CONJ which given some specific command prove that it is monotonic and conjunctive, respectively.

3.5 Calculating weakest preconditions

It is desirable to have means for calculating weakest preconditions efficiently, since this task is the first step away from the semantics towards program analysis and development. Calculating weakest preconditions can be used to synthesize commands [7, 12], and to obtain certain derived Hoare-like rules (see below).

For each command c a conversion, called a *command conversion*, is defined which given a term of the form "c q v" where q is a postcondition and v is a tuple of program variables, returns the boolean term that corresponds to the weakest precondition applied to v. It is advantageous to work with boolean logic terms since such can be manipulated using the theorems and tools available with the HOL system (conversions, rules and tactics). From now on we will also call these terms for weakest preconditions.

In order to provide an example, the conversion for assignment is called ASSIGN_CONV and is used as follows

```
#ASSIGN_CONV
#"assign
# (λ(x,y,z). x+y,y,T)
# (λ(x,y,z). y<x /\ z) (x,y,z)";;
|- ... = y<(x+y) /\ T
```

where the '...' in the theorem result stands for the input term.

A conversion called CMD_CONV has been derived from the other command conversions. It reduces any command applied to a postcondition and a tuple of program variables, to the boolean weakest precondition. It will do this recursively, until none of the commands occur in the result.

4 Derived Hoare-Like Rules

The Hoare-like rules support a forward style of program development. Starting from the basic rules (axioms), commands can be shown to meet their total correctness specifications using other Hoare-like rules. Our rules are not real Hoare rules. First of all, Hoare's approach is axiomatic, i.e. axioms and rules are postulated. Our rules are derived inferences rules in HOL which formally prove their results from the semantics of commands and total correctness specifications. This ensures total correctness specifications become logical consequences of the semantics of commands. Secondly, Hoare logic is for deducing partial correctness specifications of program whereas we consider total correctness specifications.

Total correctness specifications are defined as follows

```
⊢ ∀p c q. tc(p,c,q) = ew(p imp (c q)).
```

Thus, a command c is said to be (totally) correct w.r.t. a precondition p and a postcondition q if whenever execution of c is started in a state satisfying p, it terminates in a state satisfying q.

4.1 Pre- and Postcondition Rules

There are a number of general rules for manipulating pre- and postconditions of tc specifications. They do not depend on any particular commands but can be used for all commands. However, a couple of rules require that commands are monotonic, and one requires that commands are conjunctive. Such facts are proved automatically using the functions of section 3.4.

4.2 Basic Rules

The command conversions of section 3.5 are used to obtain Hoare-like rules without hypotheses, i.e. axioms, which give ways of reasoning about the basic commands. The basic rules are obtained as the composition of an inference rule for introducing total correctness specifications from equalities and a command

conversion. The total correctness rule can be specified as follows

```
TC_RULE : thm -> thm

|- c q v = P
------------------
|- tc(p,c,q)
```

where the variables `v` should be free in the assumptions of the hypothesis of the rule, and `p` is "λv.P" or the η-reduced version of this term (precisely when `P` is of the form "p v"). The conclusion follows by definition of total correctness specifications.

Recalling the command conversions take terms of the form "c q v" and return theorems of the form $\vdash$ `c q v = P` , we obtain the basic Hoare-like rules for skip and assignment, as follows

```
let SKIP_RULE = TC_RULE o SKIP_CONV;;

let ASSIGN_RULE = TC_RULE o ASSIGN_CONV;;
```

Of course, the same strategy can be used to obtain similar rules for the other commands. However, this is not convenient for recursively defined commands as, for instance, sequential composition and iteration since preconditions will contain commands and also in other ways be to complicated. For these, Hoare-like rules are described below. However, in some cases the rule

```
let CMD_RULE = TC_RULE o CMD_CONV;;
```

might be feasible. Given any command, postcondition and tuple of variables, it will calculate a total correctness specification where the (weakest) precondition contains no command constructs.

4.3 Blocks

The derived Hoare-like rule for the block command can be specified as follows

```
BLOCK_RULE : term -> thm -> thm

|- tc((λu.[p v]),c,(λu.[q v]))
-------------------------------------
|- tc(p,block (λw.λv.u) (λu.v) c,q)
```

The term parameter which should be a tuple of program variables `v` , is used to determine the state space of the block construct. They must be among the variables in `u` .

4.4 Sequences and Conditionals

The rule for sequential composition is straightforward. Given two hypotheses $\vdash$ `tc(p,c1,q)` and $\vdash$ `tc(q,c2,r)` it enables the deduction $\vdash$ `tc(p,c1 seq c2,r)` provided `c1` is monotonic.

The rule for conditional composition is not based on properties of commands. It has three hypotheses, of which two require each of the guarded commands establishes the postcondition if they are executed, and one require at least one of the guards is satisfied if the precondition is.

4.5 Iteration

The rule for iteration is based on the iteration theorem which provides a way of avoiding the complicated definition of the do-command in program verification situations. This is a complex theorem which gives an inductive way of reasoning about the do-command using well-founded sets. A theory of well-founded sets was therefore formalized.

A well-founded set is set with a relation such that all non-empty subsets have a minimal element w.r.t. the relation. This is expressed in the defining theorem

```
⊢ ∀C R.
    wfs(C,R) =
    (∀A. ~(A = EMPTY) /\ A SUBSET C ==>
         (∃x. x min_e (A,R)))
```

where the notion of minimal element is defined by

```
⊢ ∀x A R.
    x min_e (A,R) =
    x IN A /\ ~(∃y. R y x /\ y IN A)
```

and the constants `EMPTY` and `SUBSET` (and `IN` used above) can be interpreted as their names indicate. They are defined in the `all_sets` library of the HOL system.

We have proved two main theorems about well-founded sets. One theorem states that a set with an relation is well-founded if and only if there are no infinite decreasing chains consisting of elements in the set. Or equivalently, all decreasing chains are finite. The other theorem states that a set is well-founded if and only if it is possible to do mathematical (well-founded) induction over the set in order to prove some predicate holds for all elements.

In addition, we have proved a number of constructions on well-founded sets. These state that sets (or pairs of sets) with the subset relation, the product relation, the lexicographical relation and the inverse image relation inherit the well-foundedness of sets from which they are constructed.

The following theorem is called the iteration theorem

```
⊢ ∀C R c p b t.
    monotonic c /\
    wfs(C,R) /\
    ew((p and b) imp (λv. (t v) IN C)) /\
    (∀x.
      x IN C ==>
      tc(p and (b and (λv. t v = x)),c
         p and (λv. R(t v)x))) ==>
    tc(p,do b c,p and (not b))
```

Given predicates `p` and `b` and a command `c` the theorem states conditions which are sufficient to ensure that the truth of `p` is preserved by execution of the iteration command `"do b c"`, i.e. if `p` holds prior to execution of this loop construct then execution will terminate and `p` will hold upon termination. The monotonicity assumption about the subcommand `c` is necessary in order to ensure the term `"do b c"` actually denotes the desired command, i.e. to ensure the existence of an extreme solution.

The iteration rule, called `DO_RULE`, just implements the use of this theorem. It has three hypotheses corresponding to the second, third and fourth assumptions — remember monotonicity of any valid command can be proved using `CMDS_MONO`. Special versions of the rule have been defined for various examples of well-founded sets, for instance, the positive natural numbers, the positive integers and the natural numbers with the lexicographical relation.

5 Generating Verification Conditions

On top of the derived Hoare-like rules, we implement a collection of tactics. These make it possible to use the rules in a backwards way, and thus, provide tools for obtaining verification conditions from total correctness goals interactively.

The tactics constitute the basis of the verification condition generator. However, various information is needed in order to treat iteration and, in certain cases, sequential composition. Using tactics interactively, such information can be supplied via parameters of tactics. However, the vc generator should produce vc's automatically, so we need to implement some way of providing the tactics with arguments. Usually this is done by annotating specifications with the information where it is needed, i.e. by writing the needed information at an appropriate place in the program text. Instead, we will annotate specifications with labels — we call this labelling specifications. This means that we will just put a mark at an appropriate place which can be used by the vc generator to collect the needed information from a list argument. Thus, specifications are separated more sharply from their proofs, and specifications and programs become more readable.

5.1 Definition of Labelling

In order to provide a mechanism for labelling commands, we introduce a new constant `"label:* -> ptrans -> ptrans"`. So, we allow any HOL term to be used as a label (due to the type `":*"`). The defining theorem of `label` is

```
⊢ ∀ll c. label ll c = c
```

which says that labelling a command has no effect on the behaviour of the command.

5.2 Labelling Sequential Composition

The usual tactic for sequential composition is not capable of exploiting labels in sequences of commands. Instead it expects that the predicate needed to break apart a sequence is given via a parameter. Therefore we can define a new tactic, called `LABEL_SEQ_TAC`, on top of the old one which once it has obtained the needed predicate using the label and a list of labels and predicates, employs the old tactic on the goal (after the label has been removed). In addition, we will build a new feature into this tactic which makes it possible *not* to label commands, anyway. Instead the tactic will calculate the needed predicate using command conversions as `ASSIGN_CONV`, `SKIP_CONV` and even `CMD_CONV`. For instance, assert and assignment command never need to have a label whereas it is sometimes convenient to label conditional composition and iteration should always be labelled.

5.3 Labelling Iteration

Three pieces of information are needed to reduce a total correctness specification for a do-command, namely, a theorem specifying a well-founded set, an invariant and a bound function. We therefore implement a tactic for specifications of do-loops, called

LABEL_DO_TAC, which is able to use labels for obtaining the needed information. The first idea one gets is to implement the tactic on top of DO_TAC, the tactic derived from DO_RULE, such that the needed information is obtained as in LABEL_SEQ_TAC. However, this does not work here since special versions of DO_TAC have been derived for particular well-founded sets, and we even might want to derive new ones. It should also be possible to use such tactics. Therefore, we will associate tactics with labels, instead of invariants, etc.. These tactics must be ML functions of type `term -> tactic` which given a tuple of program variables can reduce the specifications of do-commands. Thus, the three pieces of information mentioned above are only associated with labels, implicitly, via the tactics.

5.4 The Verification Condition Generator

All that the verification condition generator must do is just to find out which tactic to apply to a given goal, — if the command is an assignment then apply ASSIGN_TAC, if the command is sequential composition then apply LABEL_SEQ_TAC, etc., — and to keep track of what the current tuples of program variables are. And, of course, then it should work recursively on subgoals until only verification conditions are left on the goal stack. It is necessary to keep track of program variables because the set of program variables changes when a specification involving a block command is reduced.

The name and functionality of the vc generator can be described as follows

```
VC_TAC: (term # term) list ->
        (term # (term -> tactic)) list ->
        term ->
        tactic
```

As it appears VC_TAC takes three arguments before it returns a function of the ML type `tactic`. The first parameter of the function is used to break apart sequential composition. The list is passed on to LABEL_SEQ_TAC and thus, it must consist of pairs of a label and a predicate. The second parameter is used to reduce do-loops and is passed on to LABEL_DO_TAC. The list should therefore consist of pairs of a label and a tactic knowing a well-founded set, an invariant and a bound function of a do-loop (which is labelled by the label component of the pair).

6 Simplification

A verification condition is usually an implication where the consequent is a minor modification of the antecedent, typically due to assignments. Proofs of vc's can be automated in part because many involve similar proof steps. For the proofs theorems about application domain theories are needed. These must be proved separately from the proofs of programs but fortunately it seems that proofs of programs share many such theorems (cf. section 7).

The basic actions of the simplification tools is to break apart vc's into manageable pieces firstly, and then to reduce conclusions of goals towards the assumptions or to derive new assumed facts from the assumptions until a proof is accomplished. We present a collection of tactics for performing such actions and for finishing off goals. The tactics are intended to be combined into more powerful proof procedures, called simplifiers. We present two examples of such which have been used to verify a number of programs in part.

The simplification tools (covering both tactics and simplifiers) are not based on decision procedures or other fancy algorithms. Instead, we try to exploit features of the HOL system. One feature is the possibility to save and reuse theorems once proved. Using such pre-proved theorems makes proofs more efficient since the number of proof steps is reduced. Besides, we use them to guide simplifications — non-trivial proof steps are always based on the use of pre-proved theorems provided by the user, or on the use of assumptions. Assumptions are used by the resolution tools of the HOL system. However, for efficiency reasons, we have implemented variants of these which use various heuristics in order to restrict acceptable resolvents. For instance, resolution tactics intended to be used recursively (but which of course can be used interactively) have been implemented which use hints and calculate new hints for subgoals. Rewriting is not used by the simplifiers (but sometimes interactively) since it would be too inefficient. Often rewriting applies in a proof but does not lead to simpler subgoals; traversing terms and proving terms equal as rewriting does, are too time consuming as tasks to just try out without knowing of the effect.

6.1 Initial Simplification

A verification condition is typically a complex term of the form

```
"∀(x1,...,xn).
  P1 /\ ... /\ Pm ==>
```

```
Q1 /\ ... /\ Qk",
```

where the P's and the Q's are arbitrary boolean terms, and the x's are program variables. An obvious way to start out the proof of a vc is therefore to break it apart into manageable pieces. Each piece will then be a new subgoal and the entire vc will follow from the proofs of the independent pieces. Once the tupled universal quantification has been removed the built-in tactic **STRIP_TAC** can be used (repeatedly) for this. Each application of this tactic removes one outer connective from the goal, either universal quantification (not tupled), conjunction, negation or implication. Goals which are disjunctions or existential quantifications must be handled interactively.

6.2 Useful tactics

After the initial simplification of a vc we are left with a large number of subgoals. Let us call these 'vc subgoals', or just 'vc goals'. Next, reasonably efficient tactics should be applied to get rid of the simplest vc subgoals. One efficient tactic is **ASM_ACCEPT_TAC** which uses the assumptions to accomplish a goal. It searches the assumption list for a term equal to the conclusion of the goal upto α-conversion, or equal to false. Another tactic is **ASM_CONTR_TAC** which tries to derive a contradiction from the assumptions, searching for a term and its negation. Finally, an efficient tactic is **MATCH_ACCEPT_TAC** which tries to finish off goals proving they are an instance of a pre-proved fact (provided as an argument). All three tactics fail if they do not accomplish a goal.

The tactic **MATCH_ACCEPT_HYP_TAC** is a variant of **MATCH_ACCEPT_TAC** which allows theorems to have hypotheses. It works on theorems of the form

```
⊢ h1 /\ ... /\ hn ==> t
```

where **n** is greater then zero, and tries to match the consequent of the theorem against the conclusion of a goal. If the match succeeds instances of the hypotheses become the new conclusions of subgoals (one for each hypothesis). Note this provides an efficient alternative to usual (unconditional) and conditional rewriting which allows the rewrite reduction to be used on top terms only. A rewrite theorem

```
⊢ h1 /\ ... /\ hn ==> (t1 = t2)
```

where **t1** and **t2** are boolean terms and **n** might be zero, can be used to accept goals under hypotheses transforming it to

```
⊢ h1 /\ ... /\ hn /\ t2 ==> t1.
```

Sometimes there is an equality among the assumptions, or an equality is obtained by resolution. We employ the heuristic that such equalities should be used for substitution in both the conclusion and assumptions of a vc goal. Note this is quite different than advocating rewriting which employs equalities with no relation to the goal, and is much more inefficient. The tactic called **ASM_SUBST_CONCL_TAC** has been implemented to substitute equalities found among the assumptions in the conclusion of the goal, whereas **ASM_SUBST_ALL_TAC** affects both the conclusion and assumptions.

6.3 Resolution Tactics

Two variants of the usual HOL resolution tactics **IMP_RES_TAC** and **RES_TAC** have been implemented. Instead of adding all resolvents to the assumptions they 1) ignore (intermediate) implicative results, 2) resolve universally quantified results recursively, 3) remove the assumptions of a goal used in the resolution, and 4) use equalities for substitution in the conclusion and assumptions of a goal before they are added to the assumptions.

However, sometimes it is desirable to restrict resolution even more than these two tactics do. In order to accomplish this we extend each goal with a list of terms. These are interpreted as *hints* (thus, a hint is just a term). We then get new kinds of goals and tactics. Let us call these 'hint goals', — the ML type used is **hgoal** denoting **term list # goal**, — and 'hint tactics'. The ML type of a hint tactic is

```
htactic = hgoal -> (hgoal list # proof).
```

Note, a hint tactic must return new hints (a hint list) for each subgoal it generates.

Any hint tactic can use the hints of a goal to control its actions. But different tactics may need different kinds of hints. Therefore it might not be possible to combine different hint tactics using hint tacticals. Since we are interested in controlling resolution tactics only, we take hints to always constitute a (small) subset of the assumptions. Resolution will then work in a more 'goal-oriented' fashion when we restrict acceptable resolvents to those which are derived from at least one of the hints.

A set of new hints **nh** can be calculated as follows

$$\mathbf{nh} = \mathbf{na} \cap (\mathbf{oh} \cup (\mathbf{na} - \mathbf{oa}))$$

where **oh** denotes the old hints, **oa** the old assumptions and **na** denotes the new assumptions. Thus, the new

hints are the terms which are among the new assumptions and, either among the old hints, or not among the old assumptions. Once hints have been used they are disregarded and thus not among the new assumptions.

Hint resolution tactics have been implemented which are similar to the restricted resolution tactics in their way of handling resolvents. Their names are `HINT_IMP_RES_TAC` and `HINT_RES_TAC`, respectively.

6.4 Two Example Simplifiers

We describe a non-recursive and a recursive simplifier for vc subgoals. All non-trivial deduction is done using pre-proved theorems, provided via parameters by the user.

Throughout this section we deal with three lists of theorems. The first is used to accept goals (`acc_thms`), the second to accept goals under certain hypotheses (`acc_hyp_thms`), and the third is used for resolving with the assumptions (`res_thms`). The simplifiers use each theorem, one by one. This gives a depth-first search strategy where all possible reductions are tried out until a proof is accomplished. In the second recursive tactic the search is continued until a certain search depth is reached (it can be set by the user).

For the implementation of the simplifiers we have implemented three proof strategies which either solve a goal or fail, and are derived from the tactics above. `STRATEGY1` uses `ASM_ACCEPT_TAC`, `ASM_CONTR_TAC` and `MATCH_ACCEPT_TAC` in that order. `STRATEGY2` uses `STRATEGY1` and some simple heuristics to prove disjunctive and existential quantified goals. On disjunctive goals `STRATEGY1` is applied to each disjunct. On existential quantified goals, the variable of the quantification is used to reduce the quantification and `STRATEGY1` is tried out on each conjunct of the body. `STRATEGY3` uses `STRATEGY2` and `MATCH_ACCEPT_HYP_TAC`, and is recursive.

The simplest simplifier uses resolution at most once to solve goals. Another uses resolution, recursively. Therefore they are called the non-recursive and the recursive simplifier, respectively. The simplifiers apply the tactics and proof strategies described above such that the less complex and less time consuming ones are tried out first. Then the conditional accept and resolution tactics are applied, and the basic strategies are used again to prove the reduced goals. The tactics are applied one after the other until a proof is accomplished. Otherwise the simplifiers give up, and terminate with a failure. The goal is then left unchanged.

The non-recursive simplifier is called `prove_vc_tac` and it takes three theorem lists as arguments, corresponding to the three lists introduced above. The proofs conducted by the tactic correspond to a path in the graph of figure 1. The arrows are read as 'then',

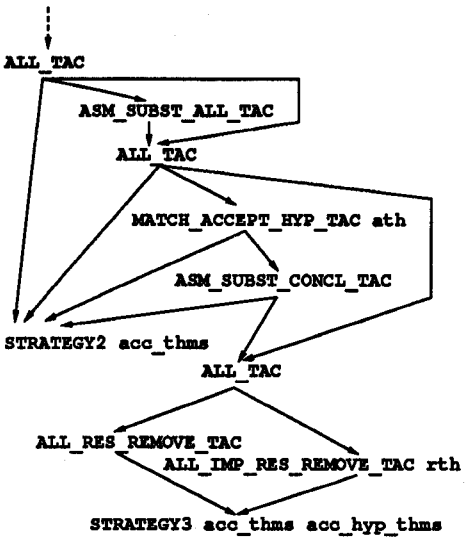

Figure 1: Proofs by the non-recursive simplifier.

and each arrow leaving a node represents an alternative for continuing a proof. Since we have not implemented heuristics to select a path, the simplifier tries out all alternatives left-to-right as they are presented in the graph.

`rec_prove_vc_tac` is the recursive simplifier and also takes the three theorem lists as arguments. In addition, it takes a number which is used as the maximum depth of proofs, and a term list used as the hint list. If the hint list is empty any first step in resolutions is allowed, and if the depth limit is zero any search depth is allowed. Proofs conducted by the recursive simplifier correspond to a path in the graph of figure 2 which is read as the graph of figure 1.

7 Examples

We have described a program verifier consisting of a verification condition generator and simplification tools, especially two simplifiers. In order to give the reader an idea about the program verifier's usability, which we measure in terms of efficiency and effectiveness, we discuss three small examples below. For simplicity, the problems and solutions are only treated informally.

Execution times have been measured on an 'empty'

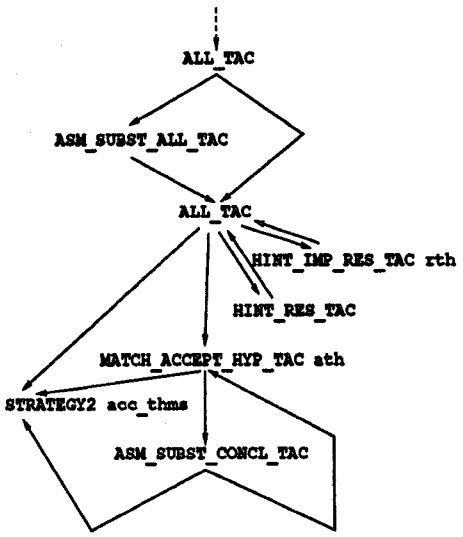

Figure 2: Proofs by the recursive simplifier.

Sun 3.60 workstation (about 1.5–2 times faster than a Sun 3.50) using the timer facilities provided with the HOL system. The times presented include both the actual run time and the garbage collection time.

7.1 Natural Number Division

We consider an algorithm for computing the dividend `d` and the remainder `r` upon dividing a natural number `p` by a positive natural number `q` [11]. The algorithm consists of a do-loop construct that contains a single multiple assignment. Starting with `r` equal to `q` and `d` equal to `0`, `r` is decreased and `d` is increased in the loop body, maintaining the invariant that `q` is positive and `p` is equal to `"r + (q * d)"`.

It is convenient to represent pre- and postconditions, invariants and bound functions by new constants in HOL. Then these constants can be used instead of the more complex terms they represent in specifications and when generating verification conditions.

In order to prove the division algorithm meets its specification, the specification is set as a goal. Then the vc generator is applied, provided with the needed arguments. This yields 5 subgoals in 22 seconds. Next, the goal is rewritten with the new constants introduced for the specification and proof. And the resulting relatively complex goals are reduced to manageable pieces using `STRIP_TAC`. These operations yield 11 subgoals and take additional 18 seconds. Now the non-recursive simplifier is applied to all eleven vc subgoals — of course this is not done interactively but the tactics are sequenced using the `THEN` tactical. The simplifier is provided with a number of basic theorems, approximately thirty divided equally among the three theorem lists. These have been selected once and forall, and consists of theorems involving various tautologies and basic facts of arithmetic which might be relevant for invariance and termination proofs of do-loops. The result is ready in 22 seconds, and there are 3 subgoals left on the goalstack. Thus, the simplifier proves 8 subgoals in 22 seconds.

The remaining subgoals were not proved for two reasons. One subgoal concerned the termination proof of the loop which is a bit different than other termination proofs, and therefore the necessary theorems were not provided with the basic theorems. After a small modification, the goal can be proved by the non-recursive simplifier provided with a couple of theorems. Here the simplifier shows its use as an interactive proof tool — just specifying the theorems needed it performs the actual proof steps. The two other subgoals required the use of rewriting with some basic facts of arithmetic, as e.g. $\vdash \forall m.\ m * 0 = 0$.

7.2 The Maximum Problem

The maximum problem can be stated as follows: for a fixed number `n>0` and a fixed array `b[0:n-1]` of numerals, write a program to compute `k` such that it is the index of a maximum element of `b[0:n-1]`. An array is represented as a function from an index type `":*i"`, here `":num"`, to some type of elements `":*"`, here also `":num"`. The notation `b[x:y]` denotes a finite subpart of array `b` consisting of the elements `b(x)`, `b(x+1)`, ..., `b(y)`.

The algorithm for solving the maximum problem (taken from [7]) is based on a traditional loop construct with an inner conditional command in order to control when to increase `k`. It needs a loop variable, say `i`, which is increased by one until the limit `n` is reached, starting from zero. Thus the bound function (also called the variant) is the difference between `n` and `i`. The invariant states that `k` and `i` must be within the right bounds, and that `k` is the index of an element greater than or equal to all other elements of `b`.

We shall do as above, defining new constants for the specification and proof of the maximum algorithm. The specification of the algorithm is set as a goal, and the vc generator is applied. This yields 7 subgoals in 50 seconds. These are rewritten using a theorem stating negation is its own inverse — double negations often arise due to do-loop guards are negated in vc's — and the new constants introduced for the spec-

ification and its proof. This yields 22 subgoals in in additional 34 seconds. Again the non-recursive simplifier provided the basic theorems is applied to the 22 subgoals. It proves 20 in 45 seconds, and thus leaves two on the goal stack. Both of these are proved by the recursive simplifier. Providing this simplifier with no hints, the goals are accomplished in 72 seconds. And with a hint on where to start the proof, it proves both goals in 18 seconds. This difference indicates the importance of the hints, but one should also remember that hints are used in recursions in both cases — even though no initial hints are provided. We cannot here go deeper into the details of the proofs but they show that the hint heuristics in recursive applications of resolutions work well in this example [1].

7.3 Bubble Sort

We consider an algorithm for arranging the elements of an array `b[0:n-1]` in non-decreasing order employing the bubble sort method [6]. It is assumed that `n` is a positive natural number and that the elements of `b` are natural numbers. The bubble sort algorithm arranges the elements in non-decreasing order by placing the largest element at the last index `n-1`, the second largest at the index `n-2`, etc. This is done by 'bubbling' ('swapping') elements to their final index one by one. It uses two nested loops. The outer loop is used to traverse the array from behind, decreasing a variable pointing to the last correctly placed element of the array `b[0:n-1]`. The inner loop is used to bubble the maximum element of the unordered section of the array to its final index.

This example differs from the others in that it is considerably more complicated. It is therefore convenient to introduce a number of constants to formalize concepts about the application domain of the algorithm. These include constants to express when a part of an array is ordered, bubble ordered (representing an intermediate state of the algorithm), among other things. The constants should not be mixed up with the constants introduced above, and also in this example, for pre- and postconditions, invariants, etc. Definitions of the latter constants are expanded before simplification is started. Definitions of the former are not since facts needed about application domain theories should be proved separately from the proof of the programs. In general, proving such facts cannot be automated.

As above, the desired specification is set as a goal and the vc generator is used to reduce it, yielding 9 subgoals in 83 seconds. Double negations and constants are removed using rewriting and vc's are reduced to simpler subgoals using `STRIP_TAC`. This yields 45 subgoals and takes 185 seconds. The non-recursive simplifier is just applied blindly to all 45, proving 30 in 236 seconds. There are two main reasons why one third of the vc subgoals are left unproven. First of all, theorems are needed about the new application domain constants. Secondly, terms of the form `"(x + 1) - 1"` make some goals more difficult than they really are — the non-recursive simplifier is not able to reduce this to `x`. Rewriting with the theorem $\vdash \forall$`m n. (m + n) - n = m` and then applying the non-recursive simplifier with additional theorems about the application domain, leaves only 2 subgoals on the goal stack and takes 236 seconds. Thus, the specification goal is reduced to 45 vc subgoals and further to 2 subgoals in 740 seconds. The reduction includes proving more than 47500 intermediate theorems. After a few initial manipulations, the final two subgoals can be proved by the non-recursive and the recursive simplifiers, respectively.

8 Conclusions

In this paper we have presented a formalization of a small imperative programming language in HOL, and a collection of tools constituting a program verifier based on the HOL system. The usability of the program verifier has been considered on a number of small examples.

The formalization of weakest precondition predicate transformer semantics seems to fulfil our goals. We have obtained a theory of general commands which is independent of types of program variables and supports the development of programs by stepwise refinement from specifications.

Representing states as tuples of values and thus, state spaces as Cartesian product spaces, provide the advantage that program variables of predicates and program can have different types. However, we also avoid semantical definitions of syntactical notions as substitution, predicate on program variables, command on program variables and refinement with respect to a fixed variable environment [11, 5]. Using tuples, the higher order logic to a large extend does the work by itself, both due to β-conversion and types of the logic, and due to the way things are expressed.

Commands are identified with weakest precondition predicate transformers in keeping with the intended semantics. Using tuples and this approach has important advantages over the representation of commands as constructors in an abstract datatype of syntax (e.g.,

used in [5]). We are able to give a more intuitive semantics of the block command (extending and restricting the state space), and the use of this command is simpler. Besides, nondeterministic and usual assignments can be defined and used in a more general and flexible way, for instance multiple assignments are possible.

There is one disadvantage of using tuples in the HOL system: λ-abstractions over tuples are handled inefficiently. Tupled λ-abstractions are not provided by the HOL logic itself, but by the parser and the pretty-printer of the HOL system (doing recursive transformations).

The program verifier consists of a verification condition generator and tools for verifying vc's. The vc generator was used on a number of small examples, and calculated vc's reasonably fast — the inefficiency of tuples affects the efficiency of the vc generator. The simplification tools also worked well saving both time and mental effort for the human verifier. The non-recursive simplifier proved between 60 and 90 percent of subgoals of vc's, automatically (given some collection of basic theorems). In the 60 percent case, more theorems about the application domain theories could be provided, increasing its performance to prove 43 of 45 subgoals (proving more than 47500 intermediate theorems in approximately 500 seconds).

We could wish for various additional features before starting out on bigger programs. For instance, the programming language should be enriched with various new commands as e.g. procedures and procedure calls (for ideas see [12]). Such new constructs would be useful for structuring both programs and proofs. More theorems should be made available about application domain theories, e.g. arrays and natural numbers, in order to support the simplifiers. In addition, an interface should be implemented allowing programs to be written in a more convenient syntax than provided by the HOL definitions of commands.

Acknowledgements

I am grateful to professor Ralph-Johan Back at Aabo Akademi University, Finland, who introduced me to this subject and acted as my supervisor. He provided the central ideas for expressing predicate transformer semantics in higher order logic (which were further developed by himself and me). I am also grateful to professor Glynn Winskel at Aarhus University, Denmark, for valuable comments on a draft of this paper. The work reported here was supported in part by the FINSOFT III program sponsored by the Technology Development Center of Finland.

References

[1] S. Agerholm, *Mechanizing Program Verification in HOL.* M.Sc. Thesis, Aarhus University, Denmark, 1991 (to be ready soon).

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

[3] R.J.R. Back, 'A Calculus of Refinements for Program Derivations'. *Acta Informatica*, Vol. 25, 1988, pp. 593–624.

[4] R.J.R. Back and J. von Wright, 'Refinement Calculus, part I: Sequential Programs'. In *REX Workshop for Refinement of Distributed Systems*, Nijmegen, The Netherlands, 1989.

[5] R.J.R. Back and J. von Wright, 'Refinement Concepts Formalized in Higher Order Logic'. *Formal Aspects of Computing*, Vol. 2, 1990, pp. 247–272.

[6] R.C. Backhouse, *Program Construction and Verification.* Prentice-Hall International, 1986.

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

[8] E.W. Dijkstra and C. Scholten, *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[9] R.W. Floyd, 'Assigning Meanings to Programs'. In J.T. Schwartz (ed.), *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, Vol. 19 (American Mathematical Society), Providence, 1967, pp. 19–32.

[10] M.J.C. Gordon, 'HOL: A Proof Generating System for Higher Order Logic'. In G. Birtwistle and P.A. Subrahmanyam (eds.), *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, 1988.

[11] M.J.C. Gordon, 'Mechanizing Programming Logics in Higher Order Logic'. In G. Birtwistle and P.A. Subrahmanyam (eds.), *Current Trends in Hardware Verification and Theorem Proving*, Springer-Verlag, 1989.

[12] D. Gries, *The Science of Programming.* Springer-Verlag, 1981.

[13] C.A.R. Hoare, 'An Axiomatic Basis for Computer Programming'. *Communications of the ACM*, Vol. 12, October 1969, pp. 576–583.