Coq Support in HAHA

Jacek Chrząszcz
University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
chrzaszc@mimuw.edu.pl

Aleksy Schubert
University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
alx@mimuw.edu.pl

Jakub Zakrzewski
University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
j.zakrzewski@mimuw.edu.pl

Abstract

HAHA is a tool that helps in teaching and learning Hoare logic. It is targeted at an introductory course on software verification. We present a set of new features of the HAHA verification environment that exploit Coq. These features are (1) generation of verification conditions in Coq so that they can be explored and proved interactively and (2) compilation of HAHA programs into CompCert certified compilation tool-chain.

With the interactive Coq proving support we obtain an interesting functionality that makes it possible to carefully examine step-by-step verification conditions and systematically discover flaws in their formulation. As a result Coq back-end serves as a kind of *specification debugger*.

2012 ACM Subject Classification Software and its engineering → Software verification, Theory of computation → Program verification

Keywords and phrases Hoare logic, program verification, Coq theorem prover, teaching

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.8

Funding This work was partially supported by the Polish NCN grant no 2013/11/B/ST6/01381.

1 Introduction

Mainstream imperative programming languages give programmers a platform in which they describe computation as a sequence of operations that transform the state of a computing machine (and interact with the outside world). The resulting programs, as produced by humans, may contain mistakes. A systematic approach to eliminate mistakes leads to an arrangement where one has to provide (at least approximately) the solution again, in a different way, and then confront the two solutions to check if they match well enough. In the mainstream software engineering this is achieved on the one hand by various requirement management frameworks and on the other hand by various testing methodologies.

Software verification techniques bring here an alternative paradigm, which is based on Hoare logic [18]. The different description of the software artefact consists here in giving explicit specification for invariant properties of states that hold between atomic instructions. The obvious advantage of this approach over testing is that a verified condition ensures correctness not limited to a finite base of available test cases, but for all allowed situations.

It is worth stressing that this possibility requires understanding of mechanisms that make it possible to generalise beyond the results of experiments that are directly available to our perception. However, this requires good command of additional theoretical constructions, which are complicated by themselves and require additional educational effort.

The experience of our faculty, most probably shared by other places, shows that typical student sees Hoare logic as tedious, boring, obscure and, above all, impractical. Therefore, it is important to counter this view by showing them the topic in an attractive modern way.

There is a wide variety of mature software verification tools (e.g. ESC/Java [10, 20], Frama-C [5, 12], Verifast [19], Microsoft VCC [13], KeY [6]) that have proven to be usable in the verification process of software projects. However their design is geared towards applications in large scale software development, rather than education, which results in automatising of various tasks at the cost of making the principles behind them difficult to understand. In particular, the base logic behind such systems, with automatic verification condition generation, handling of procedure call stack and heap, is much more complicated than the initial one designed by Hoare. This opens room for tools in which students are exposed to basic techniques only, and HAHA[1] verification environment fits into this scope.

HAHA was successfully applied in the curriculum of University of Warsaw [26]. So far it relied on the proving back-end of Z3 [15]. However, development of a Hoare logic proof with help of Z3 has limitations. In particular, the process of invariant writing is not systematic – in case the formulas are too weak or in some way incorrect, the process of problem discovery is based on guesswork. For some formulas Z3 is able to generate counterexamples, which helps in error correction. Still, the solver does not always find them.

This was our main reason to develop a verification condition generator that produces formulas in the format acceptable for an interactive theorem prover, in our case Coq. The interactive prover enables here the possibility to stepwise examine the verification condition assumptions so that complete understanding is obtained on how they are used to arrive at the prescribed conclusion. Such a stepwise examination is very similar to examination of a program operation with a debugger, but this time the main focus is not on the code of the program, but on another class of statements in an artificial language, namely specifications.

A successful application of this workflow requires two basic elements. First, the formulas should be generated in a comprehensive way that can be easily related to the point of the code they come from and students should be able to quickly start proof construction on their own. Second, most of the generated verification conditions should be discharged automatically. It is acceptable that only few of them are left for manual proof development. We show here how these goals were achieved.

In addition, HAHA can be considered as a small programming language. Therefore, it is crucial to be able to compile programs written in it. As the programs, once verified, are supposed to be highly dependable, we decided to connect compilation to a highly dependable compiler chain of CompCert [22]. As a result we obtained a basis for a miniature platform that enables development of highly dependable small programs.

The paper is structured as follows. Overview of HAHA and its features is in Section 2. Section 3 presents the translation of HAHA programs with assertions to Coq. It is followed in Section 4 by description of the Coq proof tactic to automate handling of most of the proof goals. Section 5 illustrates how Coq can be used to debug specifications. Translation of HAHA programs to CompCert languages is shown in Section 6. We conclude in Section 7.

2 Overview of HAHA

We present the features of HAHA tool and programming language by showing how a particular example procedure can be verified with its help. The code of the procedure is presented

[1] HAHA is available from the page `http://haha.mimuw.edu.pl`

```
function insert(A : ARRAY[Z], n : Z, e : Z) : ARRAY[Z]

  var j : Z
      stop : BOOLEAN
      B : ARRAY[Z]

begin
  B := A
  j := n-1
  stop := false
  while j >= 0 /\ not stop  do
  begin
    if (B[j] <= e) then begin
       stop := true
    end
    else begin
       B[j+1] := B[j]
       j := j-1
    end
  end
  B[j+1] := e
  insert := B
end
```

Figure 1 The insert procedure implemented in the HAHA programming language.

in Figure 1 and it describes the well known insert procedure that inserts an element into a sorted array, so that the array remains sorted after the operation.

The input language of HAHA is that of while programs over integers and arrays. The code of programs is packaged into functions, which serve as the basic structural component that is sufficient for code management in the small. The basic language constructs of HAHA are standard and similar to those of Pascal programming language so we omit its grammar. We only remark here that one bigger departure from the standard Pascal syntax is that we do not use semicolon to terminate statements. As a consequence one line can hold only one statement, which agrees with major coding style guidelines [17, 25]. This design choice brings some difficulties in applying standard LALR parser generators, but it has the effect of keeping the source code of programs more legible for humans. As for the semantics, we designed the language so that its mechanisms and datatypes match those supported by state of the art satisfiability solvers, e.g. Z3 [15] or CVC4 [1]. The main datatypes of the language are arbitrary precision integer numbers and unbounded arrays. This design choice makes it possible to postpone discussion on programming mistakes associated with strict keeping of the available ranges and integer bounds to other stages of instruction.

In our example the function *insert* from Figure 1 works as follows. It takes three input arguments, the array A to insert element to, the length n of its range filled with values of interest, and the value e to insert into the array. We assume that the input array is sorted and the function looks for the location into which the value e can be inserted by traversing the array in a loop from the index $n-1$ down to 0. In case the first element that is not greater than e is found, a boolean flag *stop* is set. Then the function inserts e in the found place so that the ordering of the resulting array is preserved.

The procedure cannot directly modify the input array.[2] Therefore, we have to copy its content to an array B and then operate on it. The array with inserted element e is returned as the result, which is realised by the final, typical for Pascal, assignment *insert* $:= B$.

[2] We keep input array immutable to avoid more complicated binary verification conditions and be at the same time able to refer to the input array in the postcondition.

```
predicate sorted(A : ARRAY[Z], l : Z, h : Z) =
  forall i : Z, j : Z,  l <= i /\ i <= j /\ j < h -> A[i] <= A[j]

function insert(A : ARRAY[Z], n : Z, e : Z) : ARRAY[Z]
  precondition        n >= 0                  // length is non-negative
  precondition  sort: sorted(A, 0, n)         // array is sorted
  postcondition       sorted(insert, 0, n+1)  // the result is sorted
  var j : Z
      stop : BOOLEAN
      B : ARRAY[Z]
begin
  B := A
  { sorted(B, 0, n) /\ n >= 0 }
  j := n-1
  { sorted(B, 0, n) /\ j = n-1 /\ n >= 0 }
  stop := false
  { sorted(B, 0, n) /\ j = n-1 /\ n >= 0 /\ not stop }
  while j >= 0 /\ not stop  do
    invariant 0 <= j+1 /\ j+1 <= n
    invariant onSorting: sorted(B, 0, n) /\ (j+1 < n -> sorted(B, 0, n+1))
    invariant j+1 < n -> forall k : Z, j < k /\ k <= n -> e <= B[k]
    invariant stop -> (forall k : Z, 0 <= k /\ k <= j -> B[k] < e)
    invariant j+1 < n -> B[j+1] > e
  begin
    if (B[j] <= e) then begin
       stop := true
    end
    else begin
       B[j+1] := B[j]
       { 0 < j+1 /\ j+1 <= n }
       { strongerSorting: sorted(B, 0, n+1) }
       { forall k : Z, j < k /\ k <= n -> e <= B[k] }
       { stop -> (forall k : Z, 0 <= k /\ k <= j -> B[k] <= e) }
       { B[j] > e /\ B[j+1] = B[j] }
       j := j-1
    end
  end
  { (j+1 = 0 \/ stop) /\ 0 <= j+1 /\ j+1 <= n }
  { sorted(B, 0, n) /\ (j+1 < n -> sorted(B, 0, n+1)) }
  { stop -> forall k : Z, 0 <= k /\ k <= j -> B[k] < e }
  { j+1 < n -> B[j+1] > e }
  B[j+1] := e
  { sorted(B, 0, n+1) }
  insert := B
end
```

Figure 2 The insert procedure with the specifications.

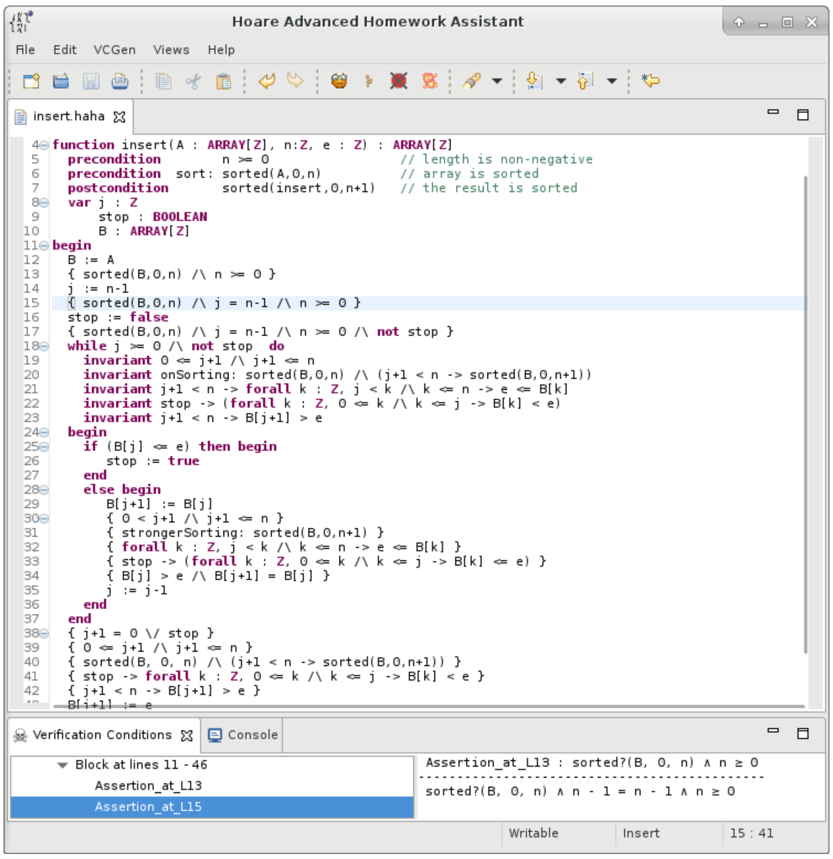
The HAHA environment does not accept directly a program written in the form presented in Figure 1. HAHA programs must contain all necessary specifications to be valid. The version of the program, which is accepted by the HAHA parser, is presented in Figure 2.

The code starts with additional definitions of predicates, which encapsulate under comprehensible identifiers essential properties that we deal with in description of the procedure states. We define there, in particular, the predicate *sorted*

```
predicate sorted(A : ARRAY[Z], l : Z, h : Z) =
   forall i : Z, j : Z,  l <= i /\ i <= j /\ j < h -> A[i] <= A[j]
```

As we can see, interpreting the expressions in natural way, the predicate holds true when the argument array A is ordered in the range $A[l],\ldots, A[h-1]$.

Another addition to the original code is a range of lines that describes the input-output property of the function. This area contains a number of properties of the input (marked with the keyword ***precondition***) and a property of the result (marked with ***postcondition***). The preconditions express that (1) the area of the values that are relevant for the procedure has at least one element, and (2) the array A is sorted in the range $A[1],\ldots,A[n-1]$. Note



```
 4 function insert(A : ARRAY[Z], n:Z, e : Z) : ARRAY[Z]
 5   precondition        n >= 0                   // length is non-negative
 6   precondition  sort: sorted(A,0,n)            // array is sorted
 7   postcondition       sorted(insert,0,n+1)     // the result is sorted
 8   var j : Z
 9       stop : BOOLEAN
10       B : ARRAY[Z]
11 begin
12   B := A
13   { sorted(B,0,n) /\ n >= 0 }
14   j := n-1
15   { sorted(B,0,n) /\ j = n-1 /\ n >= 0 }
16   stop := false
17   { sorted(B,0,n) /\ j = n-1 /\ n >= 0 /\ not stop }
18   while j >= 0 /\ not stop  do
19     invariant 0 <= j+1 /\ j+1 <= n
20     invariant onSorting: sorted(B,0,n) /\ (j+1 < n -> sorted(B,0,n+1))
21     invariant j+1 < n -> forall k : Z, j < k /\ k <= n -> e <= B[k]
22     invariant stop -> (forall k : Z, 0 <= k /\ k <= j -> B[k] < e)
23     invariant j+1 < n -> B[j+1] > e
24   begin
25     if (B[j] <= e) then begin
26        stop := true
27     end
28     else begin
29        B[j+1] := B[j]
30        { 0 < j+1 /\ j+1 <= n }
31        { strongerSorting: sorted(B,0,n+1) }
32        { forall k : Z, j < k /\ k <= n -> e <= B[k] }
33        { stop -> (forall k : Z, 0 <= k /\ k <= j -> B[k] <= e) }
34        { B[j] > e /\ B[j+1] = B[j] }
35        j := j-1
36     end
37   end
38   { j+1 = 0 \/ stop }
39   { 0 <= j+1 /\ j+1 <= n }
40   { sorted(B, 0, n) /\ (j+1 < n -> sorted(B,0,n+1)) }
41   { stop -> forall k : Z, 0 <= k /\ k <= j -> B[k] < e }
42   { j+1 < n -> B[j+1] > e }
     B[j+1] := e
```

```
Assertion_at_L13 : sorted?(B, 0, n) ∧ n ≥ 0
----------------------------------------
sorted?(B, 0, n) ∧ n - 1 = n - 1 ∧ n ≥ 0
```

Figure 3 The user interface of HAHA.

that we can name a precondition as it is the case with the second one in our code.

For educational purposes the language has an assertion mechanism that forces programmers to insert state descriptions between every two consecutive instructions of their code. We can see that the assertions are enclosed within curly brackets *{ ... }*. This makes it possible to stay very close to the original Hoare logic. In addition, programmers immediately see how big the relevant state they have to keep in mind is and observe its subtle changes along the code of their procedure. This feature forces them to explicate their understanding of programs, and as a result makes them aware of all the program details. To avoid literal repetition of assertions, we forbid assertions to occur at the beginning or at the end of a block. We can see this feature in our example by observing the lack of asserts in the loop body before the ***if*** keyword in line 25 in Figure 2. The asserts may be named as in line 31 and distributed in many statements enclosed in brackets to support structured reading of the properties (see, for example, the block of asserts between lines 30 and 34).

Loop invariants are a necessary element of any system based on Hoare logic. HAHA makes it possible to describe them through its ***invariant*** keyword, which can be used before the loop body to describe the constant property of the state at the entry to the loop, before the loop condition is checked. The textual location of the invariants is a little bit different than their reference point, but it remains in accordance with standard approach [2, 7]. Again the invariants can be named (see line 20) and understood separately, but they are combined into a conjunction when treated as a precondition for the body of the loop.

The way the procedure looks like in the user interface of HAHA is presented in Figure 3.

We can see there a window with the source code of the *insert* procedure. The editor shown there has features that are expected from a modern IDE, such as syntax highlighting, automated completion proposals and error markers. Once a program is entered, we can push one of buttons to start a verification condition generator, which implements the rules of Hoare logic. The resulting formulas are then passed to a proving back-end, which can be either Z3 or Coq, depending on the button pressed. If the solver is unable to ascertain the correctness of the program, error markers are generated to point the user to parts which could not be proven. A very useful feature of Z3 back-end is the ability to find counterexamples for incorrect assertions. These are included in error descriptions displayed by the editor.

We should remark here that to illustrate the way the Coq proof assistant can be used to debug specifications, we introduced in the specifications presented in Figure 2 a small mistake that is exploited hereafter in our explanations.

Axioms. In some cases the proving back-end is too weak to automatically handle some of the proof steps. This can be ameliorated by the use of axioms. A typical situation concerns axiomatisation of multiplication. We can for instance add an axiom

```
axiom multi: forall x : Z, (x + 1)*(x + 1) = x*x + 2*x + 1
```

which conveys some basic property of interest. Fortunately, our example program does not need axioms and actually they are not needed in case we want to use Coq as the proving back-end (although they can help to automatically prove a number of verification conditions).

3 Export to Coq

Motivations. There are several advantages of letting Coq work as the proving back-end in a tool such as HAHA. First of all, this gives the users one more option to choose, possibly one with which they can feel more comfortable. Second, SMT solvers such as Z3 always offer limited strength. This is partly mitigated by the presence of axiom construct in HAHA, but on the other hand axioms may be wrong and as a result they weaken the guarantees for the resulting program. Therefore, Coq back-end offers the possibility to prove arbitrarily difficult conditions in a way that guarantees strong confidence.

However, we would like to focus on another possibility the Coq back-end offers. One of the frequent problems in development of formal specifications in the small, i.e. in devising asserts and loop invariants within programmed functions or procedures, is that the formulas given by developers are not always good enough to carry out the proving process. There are at least three reasons for this situation. One is that the given invariant formulas are too weak to close the loop body verification effort. Another one is that a given formula has some error, which is not visible to the author. At last, the formulas may be strong enough, but they use a feature the proving back-end has difficulty to deal with (e.g. it has to make a proof by induction to exploit the information contained in the formula in a useful way).

The mentioned above weaknesses of specifications result from insufficient understanding of the algorithm details. The reason why the author gives a false specification and cannot immediately see the problem in it is that the person does not fully understand the situation in the particular location of the code. This calls for a process in which better understanding can be gained. Actually, going through an attempt to formally prove a property is a way to systematically find a gap in the understanding of the situation at hand.

Generation of verification conditions – general idea. The general idea of verification condition generation in HAHA is simple. Since asserts are given explicitly between all instructions, one has to directly apply the Hoare logic rules. However, the proof must be done in an appropriate context of defined symbols as well as predicates and the proving back-end must be supplied with this. We sketch below the main ideas behind this. The semantics of the translation is rather standard, but as the result must serve educational purposes we have to make it clear and easily accessible to the users. Therefore, we provide it here in the form close to the actual Coq text, including comments and spaces, to illustrate that the generated code can easily be related to the initial source and has a comprehensible form.

Prelude. The prelude of the Coq file contains declarations necessary to introduce the model of HAHA datatypes. We model base types of HAHA, i.e. integers (Z), and booleans ($BOOLEAN$) as Z and *bool* respectively. The combined type of arrays (***ARRAY****[τ]*) is modelled as functions from integers to the type of array elements ($Z \rightarrow \tau'$). In particular the type ***ARRAY****[BOOLEAN]* is modelled as the Coq type of functions $Z \rightarrow$ *bool*.

As a consequence the prelude in the Coq file for our example starts with library imports that introduce integer numbers to model the type Z and the domains of arrays. We also import booleans to model conditions in ***if*** and ***while*** instructions. At last, the file with our set of tactics and notations is imported.

```
Require Export ZArith ZArith.Int ZArith.Zbool.
Local Open Scope Z_scope.
Load "tactics.v".
```

The file `tactics.v` contains not only the definitions of tactics, which are described further here in Section 4, but also a basic support for our handling of booleans and arrays, i.e. a bracket notation and a bunch of lemmas that describe basic properties of the array update operation and booleans. In particular the update operation is defined there as follows.

```
Definition update {Y} (A : array[Y]) (where_ : Z) (val : Y) :=
   fun (i : Z) ⇒ if (Z.eqb i where_) then val else A[i].
```

However, the notation for update is more comfortable: the expression $A[i \leftarrow e]$ means the array A updated at the index i with the value e.

Representation of programs. The subsequent part of the file contains the module with representation of HAHA programs together with their verification conditions. The programs in HAHA are triples the first argument of which is a sequence of predicates, the second one is a sequence of axioms and the last one is a sequence of functions. Translation of a program in this form produces a Coq module called *main*. The module has the following structure

```
(** Verification context: Program correctness*)
Module main.
... Definitions of predicates ...
... Definitions of axioms ...
... Declarations of variables that represent functions ...
... Verification conditions ...
End main.
```

We would like to stress that we add to the mentioned below declarations comments that explain their relation to the original source code and verification conditions visible in the HAHA interface. We show now how the content of a Coq file sections actually looks like.

Predicates. Predicate is actually a triple of the form $(id, args, ex)$ where id is an identifier of the predicate, $args$ is its list of arguments and ex is the actual formula represented by the predicate. These elements are turned to a Coq definition as follows

```
(** Definition of the predicate id?
    – the Coq name is slightly different than in HAHA. *)
Definition id'P (args') : Prop := ex'.
```

where *id'*, *args'* and *ex'* are Coq translations of the identifier, arguments and formula.

As naming constraints in Coq and HAHA are different, the identifier in HAHA cannot be transferred to Coq with no change. Therefore, we warn the user about it in the comment. We maintain this style of comment for other definitions in the generated file. This repeated routine helps beginners a lot, while experienced users quickly learn to ignore it. In addition, the translation operation adds the suffix P to each identifier to stress that this identifier represents a predicate.[3]

Each HAHA argument of the form *id* : τ is represented in Coq by an expression of the form (*id'* : τ') where *id'* is the Coq representation of the identifier *id* and τ' is the Coq representation of the type τ. The result of translation for a sequence of such arguments is a sequence of above described Coq arguments separated with spaces.

The result of the translation for the HAHA predicate *sorted*, which we mentioned in our overview of the HAHA language in Section 2, page 4, looks as follows.

```
(** Definition of the predicate sorted?
    – the Coq name is slightly different than in HAHA. *)
Definition sortedP (A : array[Z]) (l : Z) (h : Z) : Prop :=
  forall (i : Z) (j : Z), l <= i ∧ i <= j ∧ j < h → A[i] <= A[j].
```

As we can see, the expression that defines the body of the predicate is similar to the expression in the HAHA text, which makes the whole definition easy to digest by novice users.

Axioms. Axioms differ from predicates in that they are not parametrised so an axiom is a pair of the form (id, ex) and is turned to a Coq axiom definition as follows.

```
(** Definition of the axiom id
    – the Coq name is slightly different than in HAHA. *)
Axiom id'A : ex'.
```

with meaning analogous to the one explained for the predicates. We make here one small departure by changing suffixing of the name with P to suffixing with A.

This translation applied to an example axiom written in HAHA in page 2 results in

```
(** Definition of the axiom multi
    – the Coq name is slightly different than in HAHA. *)
Axiom multiA : forall (x : Z), (x + 1) * (x + 1) = x * x + 2 * x + 1.
```

Functions. The internal structure of functions is more complicated. They are actually tuples of the form $(id, args, \tau, com, pres, posts, locs, ht)$ where *id* is the name of the function, *args* is the list of its arguments, τ is the type of the returned result, *com* is a sequence of potential comments concerning the code, *pres* is the sequence of function preconditions, *posts* is the sequence of function postconditions, *locs* is the sequence of local variable declarations, and at last *ht* is the Hoare triple that is the body of the function.

This tuple is turned into a sequence of Coq declarations as follows.

```
(** Declaration of the function name id
    – the Coq name is slightly different than in HAHA. *)
Parameter id'F : args' → τ'.
```

[3] Similar solution is adopted in Why3 [16], but there the names are prefixed. Our experience shows that it is easier to remember identifiers with suffixes that are devoted to the function than ones with prefixes so we adopted here a solution in accordance with the former choice.

```
(** Verification context: Correctness of id'*)
Module correctness_id'.
  ... translation of the body that depends on id, args, τ, pres, posts, locs, ht ...
End correctness_id'.
```

Again id' is the translation of the HAHA identifier to a Coq one with F as suffix. The arguments are translated as $args'$. This time the argument types are simply the Coq type names or their representations. Similarly, τ' is the result type name.

Before we present the details of translation for items located in the module body, let us see a concrete example for the result of this translation for our insert function *insert*:

```
(** Declaration of the function name insert
   – the Coq name is slightly different than in HAHA. *)
Parameter insertF : array[Z] → Z → Z → array[Z].

(** Verification context: Correctness of insert*)
Module correctness_insert.
...
End correctness_insert.
```

The further translation is divided into two main parts. The first one provides a representation of the local identifiers (i.e. function arguments, the function result identifier and local variables) while the second one, a representation of the verification conditions the proof of which guarantees the correctness of the source program.

The translation of local identifiers translates HAHA declarations of local variables into a series of Coq parameter constructions. For a list of local variables $locs = (id : \tau)locs'$, where id is an identifier, τ is a type and $locs'$ is a list of variable declarations, we obtain:

```
Parameter id' : τ'.
  locs''
```

where id' is a translation of the identifier id, τ' is the translation of the type τ and at last $locs''$ is the translation of the remaining identifiers.

The translation of local identifiers in the program from Figure 1 is as follows.

```
Parameter A : array[Z].
Parameter n : Z.
Parameter e : Z.
(* Variable with the function result. *)
Parameter insert : array[Z].
Parameter j : Z.
(* For boolean types we need both bool and Prop representation. *)
Parameter stop : bool.
Parameter stopP : Prop.
(* Module that ensures equivalence of stop and stopP. *)
Module StopR.
  Definition varb := stop.
  Definition varP := stopP.
  Include BoolRepresents.
End StopR.
Parameter B : array[Z].
```

One additional complexity is associated with booleans. Since these may occur both in expressions that represent values and in predicates, we need polymorphic representation for them. We decided to represent such variables as two Coq variables, in *bool* (*stop*) and one in **Prop** (*stopP*), the equivalence of which is handled by the content of the (*StopR*) module. This module, in particular, contains an axiom *(stop = true)* $\leftrightarrow$ *stopP* and some helper lemmas.

As mentioned, the second part of the function translation procedure generates verification conditions. This procedure first combines the list of preconditions into one formula that is the conjunction of the list elements. It then combines the list of postconditions into one formula using the same connective. We can now add the combined precondition formula as the assumption and the combined postcondition formula as the final condition of the mentioned above Hoare triple ht.

Hoare triples. A Hoare triple is a triple $(pres, stmt, post)$ where $pres, post$ are sequences of possibly named assertions and $stmt$ is a HAHA instruction.

The translation to Coq depends on the kind of the instruction in $stmt$. We describe translation for the block, assignment and loop instructions since these show all phenomena of interest. The translation for the remaining instructions follows the same lines.

Triples for blocks. A block is an alternating sequence of instructions and assertions. It starts and ends with an instruction. In the translation of blocks we follow the rule

$$\frac{\{\psi\}\ I\ \{\varphi\}}{\{\psi\}\ \textbf{\textit{begin}}\ I\ \textbf{\textit{end}}\ \{\varphi\}}.$$

In HAHA the situation is a little bit more complex since the body of a block is not an instruction but a sequence of instructions. As a result the Hoare triple in HAHA for which we want to define translation has the form

$$(asserts1, ((st1, asserts2) \cdot sts, st2), asserts3)$$

where $asserts1, asserts2, asserts3$ are assertions, $st1, st2$ are statements, and sts stands for the remaining, possibly empty, part of the block (concatenated with the first statement-assert pair by $\cdot$ operator). As mentioned before, the external assertions $asserts1, asserts3$ are deduced from the external context (i.e. these are external assertions, invariants, pre- or postconditions). Suppose first that sts is not empty. The result of translation starts a new module and generates conditions as follows.

```
(** Verification context: Block at lines $b_{start} - b_{end}$ *)
Module block_L$b_{start}$_$b_{end}$.
  ... translation of $asserts1, st1, asserts2$ ...
  ... translation of $asserts2, (sts, st2), asserts3$ ...
End block_L$b_{start}$_$b_{end}$.
```

where b_{start} is the first line of the block body and b_{end} is the last line of the block, and *asserts2* is the first assert in sts.

In case sts is empty the translation looks slightly differently:

```
(** Verification context: Block at lines $b_{start} - b_{end}$ *)
Module block_L$b_{start}$_$b_{end}$.
  ... translation of $asserts1, st1, asserts2$ ...
  ... translation of $asserts2, st2, asserts3$ ...
End block_L$b_{start}$_$b_{end}$.
```

To illustrate notation in the prescriptions above, we present here the actual code for module creation resulting from the block in lines 31–51 of the Figure 1.

```
(** Verification context: Block at lines 31 - 51*)
Module block_L31_51.
...
End block_L31_51.
```

Triples for assignments. An assignment is a pair (lv, e) where lv is an lvalue one can assign something to and e is an expression. Translation for assignments follows the Hoare's rule

$$\frac{\psi \implies \varphi[e/lv]}{\{\psi\}\ lv := e\ \{\varphi\}}. \tag{1}$$

Assume now that the Hoare triple for the assignment above has the form $(asserts1, (lv, e), (name, assert2))$ where $asserts1$ is the precondition, $name$ is the name of the postcondition assert and $assert2$ is the expression of the assert. The result of translation is as follows.

```
Lemma asserts1'' :
  asserts1'
  →
    assert2' .
```

The value $asserts1''$ is a name of the lemma which retrieves from $asserts1$ the line number n the assertion starts and generates the string of the form Assertion_at_L$n+1$. In case the assertion has a label, the name is simply the text of the label. The expression $asserts1'$ is the translation of the expressions in $asserts1$. The situation with $assert2'$ is more complicated. We have to construct a verification condition as prescribed in the assumption part of the rule (1). As a result, we have to construct first $assert2[e/lv]$ and the translation of this expression to Coq results in $assert2'$ above.

Consider the instruction in line 12 and the following assert in line 13 in Figure 2:

```
  B := A
  { sorted(B, 0, n) /\ n >= 0 }
```

For the assert the following lemma is generated.

```
Lemma Assertion_at_L13:
  n >= 0 ∧
  (sortedP A 0 n)
  →
    (sortedP A 0 n) ∧ n >= 0.
Proof.
  (* Give intros with meaningful names *)
  haha_solve.
Admitted.
(** Change above to Qed.
    when the lemma is proved so that
    "No more subgoals."
    occurs. *)
```

Note that as it is prescribed by the Hoare logic rule, B is replaced in the condition by A. Additionally, the instruction has no assertion that explicitly precedes it, as this is the first instruction of the function body block. In that case the preconditions of the function become the preceding assertion, in this case the conjunction of function preconditions.

One more thing we would like to bring attention to now is that aside from the verification conditions the tool generates a simple proof script. The proof script is strong enough to ascertain the validity of the conditions in most of the cases. We end the script with *Admitted* keyword and suggest to change it to *Qed* when the proof is completed. When the Coq script is finished the lemmas which end with *Admitted* give rise to error markers while those which successfully end with *Qed* lead to clean situation with no alarms in the code.[4]

Loops. The translation of the verification conditions follows here the ***while*** rule

$$\frac{\varphi \implies \theta \quad \{\theta \wedge e\}\ I\ \{\theta\} \quad \theta \wedge \neg e \implies \psi}{\{\varphi\}\ \textbf{\textit{while}}\ e\ \textbf{\textit{do}}\ I\ \textbf{\textit{end}}\ \{\psi\}}.$$

A ***while*** loop is a triple (ex, inv, b) where ex is the boolean loop guard expression, inv is the list of loop invariants and b is the Hoare triple that holds the body of the loop. If $asserts1$ is the list of asserts before the loop and $asserts2$ is the list of the asserts after the loop the resulting translation looks as follows

[4] In case *Qed* ends a reasoning which is not successful, the Coq file cannot be compiled and a message is returned to the user that verification cannot be carried out.

```
(** Verification context: Loop statement at line n*)
Module loop_Ln.
(** Verification context: Invariants before the loop*)
Module pre.
... translation of asserts1 ⟹ inv ...
End pre.
(** Verification context: Loop postcondition.*)
Module post.
... translation of inv ∧ ¬ev ⟹ asserts2 ...
End post.
(** Verification context: Invariants after a single iteration.*)
Module invariants.
... translation of the augmented body b′ ...
End invariants.
End loop_Ln.
```

where b' is b with $inv \wedge ex$ as its initial list of conditions and inv as the final one.

Conditionals. The translation of conditionals does not bring any essentially new elements, so we omit it here due to lack of space.

Expressions. The translation for equations is obvious and does not require much description. One thing we can note here is that the array operations take advantage of the function manipulation operations defined in the prelude of the Coq file, and our notations make expressions look close to their counterparts in HAHA.

4 Proof Handling in Coq

The Coq system is difficult to interact with. We mitigate the difficulties in three ways (1) by introduction of extensive comments on the Coq script, (2) by attaching an intuitive how-to instruction to each generated Coq file so that students can consult it to make proving steps of interest and (3) by development of automatic proving tactics.

The Coq system provides a wide variety of automatic verification tools, but they are not tailored to the kind of properties that result from verification condition generators. For this reason, one can significantly improve performance of proving with Coq by developing tactics adapted to the most often used patterns. We devised our own tactic instead of e.g. adopting the work of Chlipala [9] since it is important for such a tactic to be predictable, and it is difficult to control such a big tactic as the one of Chlipala with this regard.

Array updates

The first of our tactics simplifies lookups in updated arrays $A[i \leftarrow v]\,[j]$, depending on the relation of indexes i and j. For a single update there are the following cases:

- the equation $i = j$ or disequation $i <> j$ is already in the context
- one can automatically prove (by *omega*) the equation $i = j$ or disequation $i <> j$
- none of the above.

In the first two cases the tactic simplifies the update expression (either to v or to $A[j]$) and moreover in the second case the proved equation is added to the local context for future use.

In the third case, the proof must be split into two branches, for $i = j$ and for $i <> j$. The branching operation (disjunction elimination) puts the needed premises into the proof context of both branches, so the reduction of update expressions is immediate.

Given a number of update-lookup expressions it is of course more efficient to perform non-splitting reductions first and splitting ones later and this is the case in our tactic.

This procedure is repeated until no more lookups in updated tables are present in the context. The tactic running the above procedure is *solve_updates*.

Rewriting with local hypotheses

Even though there is an automatic rewriting tactic *autorewrite* in Coq, this tactic does not use equations from local context, such as premises of the theorem to be proved. Since equations between variables and other expressions are often present in verification conditions, we proposed a tactic to do exactly this.

Our tactic *simplify* takes every equation (over Z) from the local context, orients them in a way described below and rewrites the whole context with this equation using the Coq tactic **rewrite** *!H* **in** $*$.

The orientation of the equations is the following:

- equations of the form $v = exp$ are oriented from variable v to the expression, provided that v does not appear in the expression; if it does, the equation is oriented the other way around,
- if neither of the sides of the equation is a variable, the equation is oriented from the larger one towards the smaller one,
- if two sides are of the same nature (two variables, or two expressions of the same size), the equation is used from left to right.

In the first case one practically eliminates a variable from the context; even though the equation itself is not removed to help the user in understanding the proof, it is substituted by the expression everywhere else and does not add up to the complexity of the proof situation. In other cases one can hope for the decrease in the size of the proof context.

After the reduction, the *ring_simplify* tactic is called on all hypotheses and the target in order to tidy up the expressions.

The repeated rewrites (and ring simplifications) are possible thanks to the common technique of temporary hiding "used-up" hypotheses behind an identity constant, so that a given equation does not match the normal *(_ = _)* equation pattern. After the whole series of rewritings the identity is unfolded and the equations become ready to be used again.

Arithmetic forward reasoning

Although the *omega* tactic is very efficient when one wants to prove arithmetical facts, there is no support to generate new facts from existing ones. The simplest example consists in automatically generating *i=n* from the context with *i<=n* and *i>=n*. Given that the inequations may come in a variety of forms (e.g. *i<n+1*, *i−1<n*, *~i>n* etc.) the tactic is more than a simple goal matching.

We have developed an experimental tactic to perform this kind of forward reasoning. The tactic is called *deduce_equality* and its operation consists in the following steps:

1. transform all inequalities into the form *i+a* $<$ *j+b*, where i and j are variables and a, b are arbitrary expressions (hopefully integer constants),
2. given the name of the analysed variable (say i), transform the above inequalities into the form $i < j + b$ and $j + a < i$,
3. search for such a pair of inequalities that the difference $b - a$ is minimal; if it is 2 then one can conclude that $i = j + a + 1$; if it is 3 then we have two cases: $i = j + a + 1$ or $i = j + a + 2$ etc.

Currently, the tactic needs the name of the variable to concentrate on. While local variables can be matched on pretty easily, it is much more difficult to get the names of program variables, modelled in Coq as parameters. If this tactic is supposed to be used completely automatically, one has to either find the way to extract them from the environment or the Coq verification condition generator must pass their names to the tactic.

Cutting the range

It is often the case that some property, typically a loop invariant Inv, is assumed to be true for some range of arguments at hand, e.g. for $i < n$, and one has to prove that it holds for an extended range, e.g. for $i < n + 1$, typically to show that the invariant is preserved after one pass of the loop. In informal proofs one directly concentrates on the "new" part of the range, i.e. $i = n$, taking advantage of the simplifications enabled by the distilled equation.

The formal proof should first split the range into "old" and "new" part, then proof the "old" part using the assumption and leave the user with the "new" part. Unfortunately, given that there are other side-conditions to our invariant (hopefully easy to prove, but difficult to get through by automated tactics), from the Coq perspective it is only possible to know how to split the range by *trying* to do the proof. So given the goal $i < n + 1 \vdash Inv$, we apply our assumption $i < n \to Inv$ and we are left with the impossible goal $i < n + 1 \vdash i < n$ from which we can recover the information that the range should be split into the following two cases: $i < n$ and $n \le i < n + 1$.

Our tactic *limit_from_hyp* performs this kind of assumption analysis in order to split the range into "old" and "new" part. First, using existential variables and hidden *False* assumption, it sets a trap to recover some information from a following failed proof attempt. Then the tactic tries to **apply** a hypothesis to the goal. If this succeeds it means that the conclusion of the hypothesis has the same form as the goal, as in an invariant preservation proof. In that case, the premises of the applied hypothesis are combined with the current hypotheses using the arithmetic forward reasoning described in the previous paragraph, in order to recover the suitable range splitting point thanks to the prepared existential variables. At this point the tactic "backtracks" using the hidden False assumption. Then it performs the right range splitting, proves the "old" part of the range connected with the used assumption and leaves the user with "new" remaining part(s).

The tactics presented above are incorporated into a general automatic tactic *haha_solve* the invocation of which is generated as the initial body of proofs by the Coq code generator.

Even though the number of automatically proved conditions did not increase dramatically after introducing into *haha_solve* the aforementioned automatic features, these tactics are very useful when one has to resort to manual proving. Basically, using these tactics, one can prove a complex condition in a couple of lines, concentrating on higher-level reasoning instead of technicalities of the proving process.

4.1 Potential for Development

Although *haha_solve* can prove large majority of the generated verification conditions, there are still many cases which are out of reach for our automatic tactic, even though the manual proof is neither long nor particularly involved.

Naturally, there are some clear areas where a progress in automatic proving of verification conditions can and needs to be done. This includes:

1. *Integer division* Integer division, modulo etc. are operations which are not covered by good automatic tactics of Coq, yet they are quite frequent in programming and therefore in verification conditions.

Table 1 The numbers of verification conditions in example algorithms.

File	Number of VCs	Number of unproved VCs	Description
binary_search.haha	8	4	finds a value in an ordered array using bisecting approach
cubic_root.haha	9	0	computes an integer cubic root using squaring
exponent.haha	19	5	computes a power of one number to the other
heapify.haha	38	14	adds an element to a heap so that the heap structure remains intact
insert.haha	39	3	inserts an element into an ordered array (optimised version)
partition.haha	43	3	does the partition subprocedure of quicksort
sortmod2.haha	35	6	sorts elements modulo 2
square_root.haha	12	5	computes an integer square root using addition
sum.haha	7	2	finds the sum of n subsequent numbers starting with 1

2. *First order* The *firstorder* Coq tactic is not very well suited to be used as part of automatic tactics as it often takes a long time to complete (even without success) and cannot be stopped by timeout. In spite of that, elimination and introduction of existential quantification is necessary to prove verification conditions. It would also be desirable to add some support for forward reasoning with generally quantified formulas, i.e. instantiation of quantified formulas with terms at hand to facilitate the proving process.
3. *Transitivity* Sometimes proofs of inequalities require transitivity steps. In automatising such proofs one should allow for transitivity based on existential variables or guided by the facts present in the proof context. This, however, has to be allowed with caution as unlimited transitivity traversal can lead to infinite proof search.
4. *Using equations the other way around* Our automatic rewriting tactic uses some orientation of equalities in rewriting. However, especially in case of expression to expression rewriting, a possibility to rewrite in the opposite direction could sometimes help the proving process. Here also one has to be careful to avoid looping rewriting from l to r and from r to l back again.

4.2 Efficiency of the Tactics

We have run our translation to Coq procedure on a number of publicly available HAHA programs that are downloadable from the HAHA web page and that were previously fully verified with Z3 proving back-end.[5] The examples span a variety of different kinds of algorithms so they can give a reasonable impression on the strength of the tactic. Table 1 shows the number of total goals and the number of goals that were not discharged automatically. As we can see the number of the unproved goals is almost always below 10, which means the

[5] The example programs and generated Coq files can be downloaded from `http://haha.mimuw.edu.pl/coqexamplex.tgz`.

number of cases to be handled interactively is reasonably small. Moreover, out of the 210 totally generated conditions only 20% (42 goals) were not automatically proved. Of course there is a big room for improvement, but at least we have reached the satisfactory level of automatic handling.

5 Specification Debugging Using Coq

To show the support Coq can give in debugging of the specifications, we start working with the *insert* algorithm and its specifications presented in Figure 2. As we mentioned, the specifications in the figure contain mistakes. Let us try to discover it using Coq.

The first step is to generate verification conditions in Coq and see which of them cannot be proved automatically by our tactic *haha_solve*. The tactic leaves six assertions to be solved, namely *Assertion_at_L38*, in *Loop postcondition* section, *Invariant_at_L22_1*, related to the invariant correctness for the invariant in line 22 in case the first branch of the conditional is taken, from *Invariants after a single iteration* section, *Invariant_at_L21_2*, *Invariant_at_L22_2*, *Invariant_at_L23_2*, related to the invariant correctness for the invariant in lines 21, 22 and 23 in case the second branch of the conditional is taken, located in *Invariants after a single iteration* section, and *Assertion_at_L44* after the loop sections.

The correctness of *Assertion_at_L38* can be relatively quickly established by simple logical transformations. Also simple case analysis followed by application of the *haha_solve* tactic resolves goals *Invariant_at_L21_2*, *Invariant_at_L22_2*, *Invariant_at_L23_2*, as well as *Assertion_at_L44*. A different situation is for *Assertion_at_L22_1*.

```
Lemma Invariant_at_L22_1:
  0 <= j + 1 ∧  j + 1 <= n ∧
  (sortedP B 0 n) ∧ (j + 1 < n → (sortedP B 0 (n + 1))) ∧
  (j + 1 < n → forall (k : Z), j < k ∧ k <= n → e <= B[k]) ∧
  (stopP → forall (k : Z), 0 <= k ∧ k <= j → B[k] < e) ∧
  (j + 1 < n → B[j + 1] > e) ∧
  j >= 0 ∧ ~ stopP ∧
  B[j] <= e
  →
    True → forall (k : Z), 0 <= k ∧ k <= j → B[k] < e.
Proof.
  ...
```

This form of the verification condition is difficult to digest so we have to decompose it into smaller pieces. This is done with help of **intros** and **decompose** [**and**] *H* tactics. The resulting proof state is as follows

```
1 subgoal
H0 : True
k : Z
H1 : 0 <= k <= j
H2 : 0 <= j + 1
H4 : j + 1 <= n
H3 : sortedP B 0 n
H5 : j + 1 < n → sortedP B 0 (n + 1)
H6 : j + 1 < n → forall k : Z, j < k <= n → e <= B [k]
H7 : stopP → forall k : Z, 0 <= k <= j → B [k] < e
H8 : j + 1 < n → B [j + 1] > e
H9 : j >= 0
H10 : ~ stopP
H12 : B [j] <= e
______________________________________________(1/1)
B [k] < e
```

We can now see that the only way to obtain the inequality $B\,[k] < e$ is by using the hypothesis *H7* or by strengthening of *H12*, or by some kind of contradiction. We immediately see that *H7* is not usable, as it is guarded by *stopP*, which does not hold by assumption *H10*. We can now try to see if the assumption $B\ [j]\ <= e$ cannot be made stronger. Indeed, if we forcibly strengthen the condition using Coq **assert** tactic:

```
assert (B[j] < e) by admit.
```

then the proof indeed can be completed in two more steps

```
assert (B[k] <= B[j]) by haha_solve.
haha_solve.
```

as by sortedness of B the condition $B[j]\ < e$ implies the general case. This suggests to take a closer look at the situation for j. We can see that many assumptions are under the condition $j\ + 1 < n$. As our conclusion must hold also when $j{+}1 = n$, we can clear all of them to see

```
1 subgoal
H0 : True
k : Z
H1 : 0 <= k <= j
H2 : 0 <= j + 1
H4 : j + 1 <= n
H3 : sortedP B 0 n
H9 : j >= 0
H12 : B [j] <= e
_______________________________________(1/1)
B [k] < e
```

As the assumptions do not bring any information on e, except the one in *H12*, we can easily construct a counterexample by letting $B[j]\ = e$. Consequently, the assumptions are not contradictory.

This makes us conclude that the original invariant in line 22 is not provable. Clearly, the obstacle in the proof was the situation that the information in assumption *H12* used $<=$ instead of $<$. Moreover, the information is present in the assumption due to the condition in the **if** instruction. Therefore, a reasonable solution would be to weaken the invariant in line 22 to conclude with $B\ [k]\ <= e$, which indeed leads to a proper invariant.

As we can see from the case above, Coq can be used to systematically examine the situation in a particular point of the code and analyse step-by-step different circumstances that can occur there. This stepwise examination of situations is very similar to stepwise examination of the state in a debugger. Moreover, the possibility to temporarily assume new hypothesis makes it possible to change the internal state of the computation, which is similar to the state update operation available in debuggers.

6 Compilation Using CompCert

In order to provide an execution path for HAHA programs, a compiler front-end has been developed. As firm believers in practising what we preach, we decided that the compiler itself should be formally verified. As such, the front-end is built upon the infrastructure of the CompCert certified compiler [22], and is itself mostly written and verified in Coq, though it uses a trusted parser and typechecker written in OCaml. It translates HAHA into the Cminor intermediate language, which is then compiled to machine code.

6.1 Verified Compilation

By formal verification of a compiler, we understand proving that, given source program S and output code C, some correctness property $Prop(S, C)$ holds. As Leroy [21] points out, many possible properties could be described as “correctness”, e.g.

1. “S and C are observationally equivalent”;

2. "if S has well-defined semantics (does not go wrong), then S and C are observationally equivalent";
3. "if S is type- and memory-safe, then so is C".
4. "C is type- and memory-safe"

The CompCert compiler uses definition 2, and this is also the definition used in the development of the HAHA front-end.

By the definition above, we consider a compiler *Comp* to be verified correct if the following theorem has been formally proved.

$$\forall S, C, Comp(S) = \texttt{Some}(C) \to Prop(S, C)$$

Or to put it in plain English: if the compiler terminates and produces code in the target language, then this output program is observationally equivalent with the input program. A striking consequence of this definition is that a compiler that never produces any output ($Comp(S) = \texttt{None}$) is trivially considered correct. Indeed, proofs described here only provide for freedom from miscompilation issues. Bugs are possible, though they will be detected at run-time and incorrect code is never produced. Leroy [21] considers this a quality of implementation issue, to be addressed by proper testing.

In more concrete terms, the correctness theorem for the front-end is stated as follows:

$$\begin{aligned}
&\forall S, C, Comp_{\texttt{HAHA}\to\texttt{Cminor}}(S) = Some(C) \to \\
&\qquad (\forall t, r, terminates_{\texttt{HAHA}}(S, t, r) \to terminates_{\texttt{Cminor}}(C, t, r)) \\
&\qquad \wedge (\forall T, diverges_{\texttt{HAHA}}(S, T) \to diverges_{\texttt{Cminor}}(C, T))
\end{aligned}$$

Where S is the source HAHA program, C is the output Cminor program, t and T are execution traces (terminating and non-terminating, respectively), and r is the program result. In other words, we prove that the front-end, given a HAHA program, may output only those Cminor programs that produce exactly the same execution traces and results as the input program.

6.2 Specification of the HAHA Language

The first step in the process of software verification is providing a formal specification. In the case of a compiler this means formalising the semantics of the input and target languages. To this end, CompCert includes Coq definitions of the semantics for all the languages involved – C99, the target assembly languages and the intermediate languages. The HAHA front-end also provides similar specifications for its own input and intermediate languages, building upon the foundations laid by CompCert.

Dynamic semantics. Unlike recent versions of CompCert, the HAHA front-end relies on big-step (natural) semantics for the specification of languages involved. As noted by Leroy and Grall [24], big-step semantics are more convenient for the purpose of proving the correctness of program transformations, but are at a disadvantage when it comes to describing concurrency and unstructured control flow constructs [23]. The HAHA language does not have such problematic features, however, and we have decided that big-step semantics are sufficient to describe the language and specify correctness theorems in our case.

The specification reuses many elements of CompCert's infrastructure, like execution *traces* and *global environments*. Detailed description of CompCert internals is beyond the scope of this paper, however. For a detailed exposition, see [22].

The semantics is built upon the following definitions:

$$\begin{array}{lcll}
v & ::= & \mathtt{hhz}(n \in \mathbb{Z}) \,|\, \mathtt{bool}(b \in \{true, false\}) & \\
 & | & \mathtt{array}(a \in \mathbb{Z} \to v) & \\
 & | & \mathtt{int}(n \in \mathbb{Z}, -2^{31} \leq n < 2^{31}) \,|\, \mathtt{undef} & \text{values} \\
t & ::= & \epsilon \,|\, \mathtt{cons}(\mathcal{E}, t) & \text{finite trace} \\
T & ::= & \mathtt{cons}(\mathcal{E}, T) & \text{infinite trace (coinductive)}
\end{array}$$

Values range over true integers, 32-bit machine integers, Booleans, and arrays. Values `undef` appear only to mark invalid arithmetic operations and should never appear in well-typed programs. Values of variables are never undefined; a variable always has a default initial value, depending on its type (either zero, false or an array of default values).

Notice that arrays are specified as immutable mappings from integers to values and there is no concept of a memory state involved. This seems surprising at first, given that the assignment and LValue syntax suggests mutable data structures. Nevertheless, this choice was made to simplify the specification of function argument passing semantics, which are by-value. Otherwise, every array would have to be copied before being passed as an argument, which would have been costly at runtime and clunky in implementation. Immutable arrays eliminate this problem, though at the same time they generate some, more manageable in our opinion, complexity in semantics of assignment statements.

Traces record details of input/output events, such as system calls or volatile loads and stores.

The following semantic judgements are defined using inductive predicates in Coq:

$$\begin{array}{lcll}
G, E & \vdash & a \Rightarrow v, t & \text{terminating expressions} \\
G & \vdash & s, E \Rightarrow out, E', t & \text{terminating statements} \\
G & \vdash & E, a := v \Rightarrow out, E', t & \text{assignments} \\
G & \vdash & fn(\vec{v}) \Rightarrow v, t & \text{terminating calls} \\
 & \vdash & prog \Rightarrow v, t & \text{terminating programs}
\end{array}$$

Expressions can produce a trace t, i.e. generate side effects in the form of function calls. A separate rule is used to specify the manner assignments affect the local environment, a nontrivial matter due to having to emulate destructive updates on otherwise immutable arrays. The semantics of HAHA statements and expressions do not make explicit use of CompCert memory states. All the state information is contained in environments E and values themselves. The global environment G contains information about functions defined in the translation unit.

At the same time, the usual inductive definitions of natural semantics are not sufficient to describe non-terminating (diverging) programs. In order to cover such cases, we use the approach proposed first by Cousot and Cousot [11], and later implemented in CompCert, which complements the ordinary inductive big-step semantics with co-inductive rules describing non-terminating evaluations.

$$\begin{array}{lcll}
G, E & \vdash & a \overset{\infty}{\Rightarrow} T & \text{diverging expressions} \\
G, E & \vdash & s \overset{\infty}{\Rightarrow} T & \text{diverging statements} \\
G & \vdash & fn(\vec{v}) \overset{\infty}{\Rightarrow} T & \text{diverging calls} \\
 & \vdash & prog \overset{\infty}{\Rightarrow} T & \text{diverging programs}
\end{array}$$

$$\frac{G, E \vdash e_1 \Rightarrow \mathtt{array}(a), t_1 \qquad G, E \vdash e_2 \Rightarrow \mathtt{hhz}(i), t_2 \qquad a(i) = v}{G, E \vdash e_1[e_2] \Rightarrow v, t_1 \cdot t_2}$$

$$\frac{G, E \vdash \vec{e} \Rightarrow \vec{v}, t_1 \qquad G(id) = fd \qquad G \vdash fd(\vec{v}) \Rightarrow v, t_2}{G, E \vdash id(\vec{e}) \Rightarrow v, t_1 \cdot t_2}$$

$$\frac{G, E \vdash \vec{e} \Rightarrow \vec{v}, t_1 \qquad G \vdash E, id := v \Rightarrow E', t_2}{G \vdash id := e \Rightarrow o, E_2, t_1 \cdot t_2}$$

$$\frac{G \vdash s_1, E_0 \Rightarrow \mathtt{out\_normal}, E_1, t_1 \qquad G, E_1 \vdash s_2 \overset{\infty}{\Longrightarrow} T}{G, E_0 \vdash (s_1; s_2) \overset{\infty}{\Longrightarrow} t_1 \odot T}$$

$$\frac{G, E \vdash e_1 \Rightarrow \mathtt{array}(a), t_1 \qquad G, E \vdash e_2 \Rightarrow \mathtt{hhz}(i), t_2 \qquad G \vdash E, e_1 := a[i := v] \Rightarrow E', t_3}{G \vdash E, e_1[e_2] := v \Rightarrow E', t_1 \cdot t_2 \cdot t_3}$$

Figure 4 Examples of HAHA semantic rules.

6.3 The Target Language: Cminor

Cminor is the input language of CompCert back-end. It is a low-level imperative language, that has been described as stripped-down variant of C [22]. It is the lowest-level architecture independent language in the CompCert compilation chain and thus is considered to be the entry point to the back-end of the compiler [22].

Cminor has the usual structure of expressions, statements, functions, and programs. Programs are composed of function definitions and global variable declarations. For a more detailed description of the language we refer to [22].

6.4 Implementation

The HAHA front-end for CompCert has a rather conventional design. It consists of a parser, type checker, and several translation passes. These are connected with the CompCert back-end by a simple unverified compiler driver.

The proofs of semantic preservation follow the pattern described by Blazy, Dargaye, and Leroy in [3]. They proceed by induction over big-step HAHA evaluation derivation and case analysis of the last rule used. They show that the output expressions and statements evaluate to the same traces, values, and outcomes as input code, effectively simulating it. Such proofs are conducted for every front-end pass and then composed into a proof of correctness for the whole translation chain. Below, we give short descriptions of the front-end passes.

6.4.1 Parsing and Semantic Analysis

The compiler uses its own unverified GLR parser for the HAHA language. Although initially planned, reusing the Xtext-generated parser of the HAHA environment for a Coq development proved difficult. The additional complexity in defining its semantics and translation into a more usable format would overshadow the benefits of that approach.

The parser produces a raw, untyped AST from the input file. This is then passed into the semantic analyser. Aside from doing type checking, the semantic analysis pass performs several minor, but important, tasks, like string interning and generation of runtime initialisation data for arbitrary precision integer literals, which are stored as global variables in the output program.

6.4.2 Expression Simplification

The first verified pass of the front-end is removal of logic specifications, which do not affect the dynamics of program runs. As it is routine we do not describe it.

The purpose of the next pass is to replace HAHA expressions with no direct equivalents in Cminor. Specifically, function calls, true integer arithmetic, and HAHA array manipulation, all of which in the end are replaced with function calls are pulled out into separate statements. It can be thought of as a form of limited three-address code generation. Following a convention established by CompCert developers, the target language of this pass is called Haha♯minor.

The main idea of the simplification algorithm is very simple. During recursive traversal of the expression tree, every HAHA expression e, which directly results in a side effect, is replaced with a reference to a fresh temporary variable t. A statement reproducing the expression's effect and assigning the resulting value to t is then inserted just before the expression occurs in the code. Additionally, HAHA loops are turned into infinite loop, with a conditional `break` statement inside the iteration.

Temporary variables are semantically similar to HAHA local variables, but separate from them – they reside in their own environment TE. Every function can have an unlimited number of temporaries. Although not made explicit in the semantics, identifiers of temporaries should not collide with the identifiers of locals and this is enforced by runtime assertions in the later stages of compilation.

The pass is analogous to *SimplExpr* pass of the CompCert C front-end and uses similar implementation and verification techniques. The semantic preservation is proved with respect to a non-executable, relational specification, expressed using inductive predicates. The specification captures the syntactic conditions under which a Haha♯minor construct could be a valid translation of a given HAHA expression or statement, without prescribing a way in which new variables are generated. It allows a simpler proof semantic preservation, which does not have to deal with the details of implementation. Translation functions are written using monadic programming style, with a dependently typed state monad to generate fresh identifiers. Their outputs are then proved correct with respect to the specification, which is sufficient to establish correctness.

To give an example, the specification for expression is a predicate of the form:

$$a \sim s, a', \vec{id}$$

where a is a HAHA expression, s is a Haha♯minor statement that reproduces the side effects of a, a' is the translated expression and $\vec{id}$ is the set of temporary variables that are referenced in a'. Example rules are given in Figure 5. Notice that uniqueness and disjointedness of temporaries in subexpressions is specified in an abstract way.

6.4.3 Further Simplifications

The next pass translates Haha♯minor into the Hahaminor intermediate language. On a program transformation level, this pass performs two basic tasks, which account for differences in function call semantics between the two languages:

1. Explicit local variable initialisation.
2. Insertion of return statements.

$$\frac{t \in \mathit{tmps}}{\mathtt{hhz}(n) \sim \mathtt{tmp}(t) := \mathtt{alloc}(n), \mathtt{tmp}(t), \mathit{tmps}}$$

$$\frac{\mathtt{typeof}(e) = \mathtt{hhz} \quad e \sim s', e', \mathit{tmps}' \quad t \in \mathit{tmps} \quad t \notin \mathit{tmps}' \quad \mathit{tmps}' \subseteq \mathit{tmps}}{\mathit{op}_1(e) \sim (s'; \mathtt{tmp}(t) := \mathtt{op}_1(e')), \mathtt{tmp}(t), \mathit{tmps}}$$

$$\frac{\begin{array}{c}\mathtt{typeof}(e_1) \neq \mathtt{hhz} \quad \mathtt{typeof}(e_2) \neq \mathtt{hhz} \quad e_1 \sim s_1, e_1', \mathit{tmps}_1 \quad e_2 \sim s_2, e_2', \mathit{tmps}_2 \\ \mathit{tmps}_1 \cap \mathit{tmps}_2 = \emptyset \quad \mathit{tmps}_1 \subseteq \mathit{tmps} \quad \mathit{tmps}_2 \subseteq \mathit{tmps}\end{array}}{\mathit{op}_2(e_1, e_2) \sim (s_1; s_2), \mathtt{op}_2(e_1', e_2'), \mathit{tmps}}$$

Figure 5 Example rules of the expression translation specification.

HAHA and all the intermediate languages up to Haha♯minor use Pascal-like special variable to hold the return value of the function. No special `return` statement is provided. Once the execution of the function body finishes, the value of the special variable is returned to the caller. Additionally, all the local variables are automatically initialised to default values. On the other hand, in Cminor the return value at the end of the function is undefined unless an explicit return statement is provided. Local variables are initially considered undefined and need to have values explicitly assigned.

Initialising variables in the translated program is very simple: it is sufficient to prepend to the function body the appropriate assignments of default values. Providing explicit returns is equally simple. The lack of any unstructured control flow features in the HAHA language means that it is sufficient to simply append a return statement to the function body.

Aside from that, there exists another significant semantic difference between the two languages, Hahaminor deals away with the second local "temporary" environment of Haha♯minor and has only a single environment for local variables. This is an important simplification step before the final pass of Cminor generation, which has plenty of difficulties in proving semantic preservation on its own.

Proof of Correctness

Relating environments. The *Haha♯minor* intermediate language uses two local environments: one for temporary variables introduced by the expression simplification pass, and one for "old" locals. Hahaminor, on the other hand, uses only a single environment. Accordingly, one of the key issues in verification of this pass is to define sensible relation connecting both Haha♯minor environments to the resulting combined Hahaminor environment.

The matching relation $\mathit{MatchEnv}(E, \mathit{TE}, E')$ between a Haha♯minor local and temporary environments E and TE, and a Hahaminor environment E' is defined as follows, assuming that the sets of temporaries and locals are disjoint:

- For all local variables x, $E(x) = E'(x)$.
- For all Haha♯minor temporary variables t, $\mathit{TE}(t) = E'(t)$.

The disjointedness property, despite being easy to provide, is difficult to prove. Instead, a choice was made to insert runtime assertions that would enforce this property in the translation procedures. In the spirit of the definition given in Section 6.1, if the compiler contained a bug that violated this invariant, an error would be produced.

6.4.4 Cminor Code Generation

The next and final pass of the front-end translates Hahaminor into Cminor code, which can then be fed into the CompCert back-end. The translation deals with the encoding of operations of HAHA values into lower-level constructs available in Cminor.

Boolean values are turned into integers and operations on them are rewritten accordingly. Loops are put inside `blocks` and `break` statements are replaced with `exit` statements that jump outside the block of the innermost loop:

$$\texttt{break} \Rightarrow \texttt{exit}(0) \qquad \frac{s \Rightarrow s'}{\texttt{loop}(s) \Rightarrow \texttt{block}(\texttt{loop}(s'))}$$

Array operations and arbitrary precision arithmetic are lowered into runtime library calls. The pointers to initialisation data for integers are provided by the means of an axiom, that is instantiated during program extraction to a hash table lookup.

Proof of Correctness

Despite the simplicity of the code transformations in this pass, the proof of semantic preservation is rather involved. The most problematic aspect is bridging the incompatible worlds of HAHA and Cminor values.

The runtime functions which implement manipulation on HAHA values are specified to accept and return opaque pointers. A sensible preservation proof needs a way to associate those pointers with values they represent.

Following Dargaye [14], we divide memory blocks into following categories:

- Stack blocks (`SB`). At every function call, Cminor allocates a new stack block to contain the function's activation record. The block is freed upon exit from the call.
- Heap blocks ($\texttt{HB}(v)$). Pointers to these blocks represent HAHA integer and array objects.
- Global blocks (`GB`). Functions and integer constant initialisation data. A global block is allocated at the very beginning of program execution and it remains live for the entire runtime of the program.
- Invalid blocks (`INVALID`).

Let $f(M, b)$ be a mapping assigning one of those categories to memory blocks b, for a given memory state M. We may define a value matching relation, parametrised by a memory state M and the mapping f, as follows:

- $\texttt{int}(n) \approx_{(M,f)} \texttt{int}(n)$
- $\texttt{true} \approx_{(M,f)} \texttt{int}(1)$
- $\texttt{false} \approx_{(M,f)} \texttt{int}(0)$
- $\forall v, \texttt{undef} \approx_{(M,f)} v$
- $\dfrac{f(M, blk) = \texttt{HB}(\texttt{hhz}(n))}{\texttt{hhz}(n) \approx_{(M,f)} \texttt{ptr}(blk, 0)}$
- $\dfrac{f(M, blk) = \texttt{HB}(\texttt{array}(a))}{\texttt{array}(a) \approx_{(M,f)} \texttt{ptr}(blk, 0)}$

Values that are represented by pointers are manipulated using runtime library functions, whose behaviour is specified using axioms.

Relating environments. The matching relation $MatchEnv(M, f, E, E')$ that connects a Hahaminor local environment E, block mapping f, and Cminor memory state M and environment E' is defined as follows:

- For all Hahaminor variables x, $E(x) = v$ there exists Cminor value v', such that $E'(x) = v'$ and $v \approx_{M,f} tv$.
- $f \parallel M$,

where $f \parallel M$ denotes a relation that holds if for all b:

- b is not a valid block of M, iff $f(b) = \texttt{INVALID}$,
- b is a valid block of M, iff $f(b) \neq \texttt{INVALID}$.

6.4.5 Runtime Support for HAHA Programs

Like many high-level languages, HAHA has numerous features that do not map directly to features of commodity hardware or the Cminor language. Some of those features, like HAHA Booleans, can be relatively cheaply transformed into inline code. Others require complex algorithms and make use of features like dynamic memory allocation, which in practice require external libraries implementing them.

Since developing the language runtime was considered to be of secondary importance, it was decided to wrap off-the-shelf open-source components to provide most of the functionality. As such, currently the runtime system forms a trusted computing base. Still, we do not consider this a fatal blow, as the libraries we used are widely used and considered reliable.

Current implementation of arbitrary precision integer arithmetic is a thin wrapper around the *GNU Multiple Precision Arithmetic Library* (GMP).

The unusual semantics of HAHA arrays, namely unbounded size and immutability, can be efficiently implemented using functional dictionaries. Indeed, they are currently implemented using balanced binary search trees.

Memory management is currently provided by the Boehm-Demers-Weiser conservative garbage collector [4]. Given that the compiler is not intended to be used in production environments, we consider conservative collection to be adequate.

7 Conclusions and Further Work

Current version of HAHA (0.57) can be viewed as a basic verification platform for programming in the small. It allows one to write imperative procedures and their input-output specifications. Then the specifications can be interactively examined and proved in the Coq proof assistant. Our Coq scripts are systematically filled with information that makes it easy to connect the proof script with the code of the original program. A program, once verified, can be compiled through CompCert compilation chain with its behavioural guarantees. In this way, one can use a types-based tool to teach students the basics of the contemporary software verification technology.

The further steps in the development of HAHA include addition of automatic verification condition computation in the style of weakest precondition generation and introduction of function call stack. However, we are very cautious in introduction of these elements as they will inevitably make presentation of various expressions in the proving back-end complicated and likely to be less readable as it is commonly seen. One possible way to achieve this is to extend ideas used in the CFML project [8] and hide in a careful way the notational overhead introduced there to achieve a more general solution.

References

1 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proc. of CAV 2011*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

2 Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009.

3 Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *LNCS*, pages 460–475. Springer, 2006. URL: `http://gallium.inria.fr/~xleroy/publi/cfront.pdf`.

4 A garbage collector for C and C++. Retrieved November 16, 2016, from `http://www.hboehm.info/gc/`.

5 Sylvie Boldo and Claude Marché. Formal verification of numerical programs: From C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5(4):377–393, December 2011.

6 Richard Bubel and Reiner Hähnle. A Hoare-style calculus with explicit state updates. In Zoltán Instenes, editor, *Proc. of Formal Methods in Computer Science Education (FORMED)*, ENTCS, pages 49–60. Elsevier, 2008.

7 Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proc. of FMCO 2005*, pages 342–363, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

8 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN International Conference on Functional programming (ICFP)*, pages 418–430. ACM, 2011.

9 Adam Chlipala. *Certified Programming with Dependent Types.* The MIT Press, 2013.

10 David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Proc. of CASSIS 2004*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.

11 P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the Ninthteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, New Mexico, January 1992. ACM Press, New York, NY.

12 Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Proc. of SEFM'12*, volume 7504 of *LNCS*. Springer, 2012.

13 Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion*, pages 429–430. IEEE, 2009.

14 Zaynah Dargaye. *Vérification formelle d'un compilateur pour langages fonctionnels.* PhD thesis, Université Paris 7 Diderot, July 2009.

15 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

16 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proc. of ESOP'13*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

17 Google Java style guide. Retrieved November 16, 2016, from `https://google.github.io/styleguide/javaguide.html`.

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

19 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

20 K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, October 2000.

21 Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. of POPL'2006*, pages 42–54. ACM Press, 2006. URL: `http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf`.

22 Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

23 Xavier Leroy. Mechanized semantics for compiler verification. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems, 10th Asian Symposium, APLAS 2012*, volume 7705 of *LNCS*, pages 386–388. Springer, 2012. Abstract of invited talk. URL: `http://gallium.inria.fr/~xleroy/publi/mechanized-semantics-aplas-cpp-2012.pdf`.

24 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. URL: `http://gallium.inria.fr/~xleroy/publi/coindsem-journal.pdf`.

25 Linux kernel coding style. Retrieved November 16, 2016, from `https://www.kernel.org/doc/Documentation/CodingStyle`.

26 Tadeusz Sznuk and Aleksy Schubert. Tool support for teaching Hoare logic. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Proc. of SEFM 2014*, volume 8702 of *LNCS*, pages 332–346. Springer, 2014. `doi:10.1007/978-3-319-10431-7_27`.