Hoare Advanced Homework Assistant User Manual

Release 13/11

Tadeusz Sznuk, Aleksy Schubert

November 06, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

HAHA is a programming language embedded in a modern program development environment based on Eclipse. The purpose of HAHA is to teach students Hoare logic. A programmer can write simple programs and annotate them with Hoare logic assertions. The environment verifies the assertions against the code and discharges them with help of external theorem provers, both automated and interactive. A user can write programs that manipulate on true integers and on arrays.

This document describes version 0.5 of HAHA. It serves as a user manual. It contains instructions on how to start using the tool, a gentle introduction the HAHA language, a description of the outputs served by the environment, language reference, and a set of more complicated examples.

This document is not meant to serve as a developer manual. Details of the design of the tools and description of how to effectively adapt the tool to the needs of a particular formal methods course are presented elsewhere.

1.1 Availability

You can download the newest version of the program from http://www.mimuw.edu.pl/~tsznuk/haha/

1.2 Contact

You can contact the authors through the email haha@mimuw.edu.pl.

1.3 Acknowledgements

The project was partly financed by Polish government grant N N206 493138. We are grateful Andrzej Tarlecki for continuous mental support for the project.

1.4 Requirements

- Java 1.6
- Microsoft Z3
- The program is actively tested on Windows 8, Debian Wheezy and Linux Fedora 18. In principle it should also work on other Windows or Linux based operating systems.

Note: We also provide binaries for Mac OS X, but we currently lack the resources necessary to test them properly and provide support.

CHAPTER

TWO

STARTING WITH HAHA

2.1 Installing required software

2.1.1 Java

HAHA requires JRE 1.6, which can be downloaded from its website. Most Linux distribution also contain Java in their package repositories. On Debian and Ubuntu systems it can be installed with the following command:

```
sudo apt-get install openjdk-6-jre
```

Note that it is not necessary to install the Java Development Kit (JDK), only the JRE is required to run HAHA.

2.1.2 SMT solver

HAHA relies on a SMT solver to prove the validity of generated formulae. Currently the only supported solver is Microsoft Z3. It can be obtained from its Codeplex site. The download page provides stable binaries for Windows (both 32 and 64 bit), users of other systems must either

- Use a nightly build, accessible by clicking ‘Planned’ under ‘Other downloads’ on the download page.
- Or compile from source by following the procedure described below.

To compile Z3 from source, perform the following steps

- Download source code of the stable branch, either by clicking on the ‘Download’ link in the ‘Source code’ tab of the website or cloning the git repository at https://git01.codeplex.com/z3.
- Unpack the source and enter the following commands:

  ```
  autoconf
  ./configure --prefix=/usr/local
  python scripts/mk_make.py
  cd build
  make
  sudo make install
  ```

- Make sure that binaries are available on system `PATH`. The commands given here install binaries in `/usr/local/bin`, this can be modified by adjusting the `--prefix` option in the second line. In particular, it is possible to install Z3 in home directory, without root privileges.

Note: The compilation requires some additional software

- C++ compiler (obviously)
- autoconf
- make
- python

On Debian and Ubuntu systems, these can be installed with:

```
sudo apt-get install autoconf make python g++
```

Note: In Mac OS X the standard C++ compiler is `clang++` (based on LLVM), rather than `g++` (GNU C++). Mac users should append the text `CXX=clang++` to the `configure` command.

Windows users can simply install the precompiled binaries from the website. It is still necessary to ensure that installed executables are available on the system `PATH`.

2.2 Installing HAHA

HAHA is distributed as a single archive which contains builds for all supported systems. There is no installation procedure other than unpacking the directory containing the proper version from the archive. We provide both 32 (x86) and 64 (x86_64) bit variants of HAHA. It is necessary to choose the one that matches the version of Java available on the system. It can be obtained with the following command:

```
java -version
```

Startup options can be configured in the `haha.ini` file found in the installation directory. Changing these options might be desirable in the following circumstances

- In case of errors related to insufficient memory. This problem can be addressed by appending the following options to the config file
  - `-Xmx<total-memory-in-mb>m` (e.g. `-Xmx1024m`).
  - `-Xms<stack-memory-in-mb>m` (e.g. `Xms128m`).
  - `-XX:MaxPermSize=<permgen-memory-in-mb>m` (e.g. `-XX:MaxPermSize=512m`). Note that this value must be at least 256m for the IDE to work (512m is recommended).
- If Unicode characters are not displayed correctly (especially in the outline view), append the following option:

  ```
  -Dfile.encoding=UTF-8
  ```

Note: VM options must be given on separate lines, after the `-vmargs` line.

2.3 Running

To run HAHA, launch the `haha` executable from the installation directory. HAHA is typical file editor with standard and intuitve commands. One thing to keep in mind is that source files should have the extension `.haha` - toherwise the editor might not function properly.

The following sample can be pasted into the editor to verify that it is working correctly

```
function hello() : Z
  postcondition hello = 4
  hello := 2 + 2
```

The code should be highlighted (assuming that the correct file extension was used). To start the verification process, right-click anywhere in the editor are and choose `Generate VCs` from the displayed menu. This command can be also accessed from the main menu and the toolbar. HAHA will then present a console with computed verification conditions and messages logged during verification. If there were any problems, error markers will be added to the editor.

CHAPTER

THREE

GENTLE INTRODUCTION TO HAHA LANGUAGE

Suppose that we want to write a function that given a number *n* returns the sum of all subsequent numbers from *1* to *n* inclusively. We start with a header of the function:

```
function sum( n : Z ) : Z
```

it contains the keyword `function` that tells the system to interpret the following expressions as a function. Then the name of the function is given, in this case it is *sum*. The information about its parameters is enclosed in the parenteses. In our case, we have one formal parameter that is called *n*. We declare the type of the parameter after the colon. It is **Z** this time, i.e. the type of integer numbers as we know them from mathematisc (these are not 32-bit integer numbers frequently met in programming languages). In the end, the parentesis with parameters is followed by the declaration of the result type. In our case this is again **Z**.

This function header can be followed by the definition of the body. The header with the body together look as follows:

```
function sum( n : Z ) : Z
begin
     x := 1
     y := 0
     while x <= n do
     begin
             y := y + x
             x := x + 1
     end
     sum := y
end
```

The code should not be surprising as we have all seen at least one implementation of the sum in our lives, especially for those who are familiar with Pascal as much of the syntax is based upon the language. However, let us discuss the details of the definiton to feel at ease in further development of other programs. First of all, the code as it stands is not syntactically correct. We assume in our language that all variables that are used in code must be declared beforehand. Therefore, we need to add between the function header and body declarations of the variables `x` and `y` so that the code is as follows:

```
function sum( n : Z ) : Z
var x : Z
    y : Z
begin
     x := 1
     y := 0
     while x <= n do
     begin
             y := y + x
             x := x + 1
     end
    sum := y
end
```

The keyword `var`, as in Pascal, marks the beginning of variable declaration sequence. Unlike Pascal, the elements of the sequence are not separated with semicolons `;`, but with newlines. This holds both for variable declarations and instructions.

The variable assignemt is done in the Pascal style as in:

```
x := 1
```

or:

```
y := y + x
```

We also use the Pascal style to define the return result of the function, i.e.:

```
sum := y
```

The while loop is defined by a phrase of the form:

```
while x <= n do
```

followed by an instruction the loop iterates over. In our case this is a block instruction enclosed between `begin` and `end`, i.e.:

```
begin
        y := y + x
        x := x + 1
end
```

To express our intent with regard of the function we can add a precondition and postcondition formulae. These are located between the function header and its body. This looks as follows in this case:

```
function sum( n : Z ) : Z
precondition n >= 1
postcondition sum = n * (n+1) / 2
var x : Z
    y : Z
begin
...
```

In this case the precondition, introduced with `precondition` keyword, says that the function can be called when the parameter *n* is not less than *1*. The postcondition, introduced with `postcondition` keyword, says that the result is equal to the commonly known (oh, our Gauss heritage) closed formula for the sum of the integers from *1* to *n*. These conditions can be named to make future reference easier:

```
precondition natural: n >= 1
postcondition gauss: sum = n * (n+1) / 2
```

Hoare logic prescribes that each instruction must be surrounded by two assertions. The first one describes the condition of the program state that is expected before the instruction and the second one describes the state resulting from the execution of the instruction. The precondition-postcondition pair are the assertions for the whole function. The assertions for other instructions are written in curly brackets located between instructions. To save notational burden we do not write the first and last assertions in function as these are expressed with precondition and postcondition respectively. Therefore, the initial assignments decorated with the assertions look as follows:

```
begin
    x := 1
    { n >= 1 /\ x = 1 }
    y := 0
    { n >= 1 /\ x = 1 /\ y = 0 }
    while x <= n do
```

The loop invariant condition, i.e. the formula that at the entry point to the loop at each its iteration, is marked with a special keyword `invariant`. So the while loop header augmented with the invariant is as follows:

```
while x <= n do
invariant y = x * (x-1) / 2 /\ x<= n+1 /\ n >= 1
begin
```

This invariant formula can be named to make future reference more accurate:

```
invariant gauss: y = x * (x-1) / 2 /\ x<= n+1 /\ n >= 1
```

Again, the presence of the invariant formula gives us excuse not to mention assertions at the beginning and at the end of loop body as they equal to the invariant. This makes the loop body look as follows:

```
begin
    y := y + x
    { y - x = x * (x-1) / 2 /\ x<= n /\ n >= 1 }
    x := x + 1
end
```

We can now combine all the assertions with the code and obtain the complete example:

```
function sum( n : Z ) : Z
           precondition natural: n >= 1
           postcondition gauss: sum = n * (n+1) / 2
           var x : Z
               y : Z
begin
           x := 1
           { n >= 1 /\ x = 1 }
           y := 0
           { n >= 1 /\ x = 1 /\ y = 0 }
           while x <= n do
           invariant gauss: y = x * (x-1) / 2 /\ x<= n+1 /\ n >= 1
           begin
                     y := y + x
                     { y - x = x * (x-1) / 2 /\ x<= n /\ n >= 1 }
                     x := x + 1
    end
           { y = n * (n+1) / 2  }
           sum := y
end
```

CHAPTER

FOUR

RESPONSES OF HAHA

4.1 Error markers

HAHA reports errors by displaying error markers in the editor. Each marker has a tooltip which describes the reason why it was created. This information is also available in the console view (located in a tab in the bottom part of the window). This chapter describes various kinds of error markers produced by HAHA.

4.2 Syntax errors

Markers for syntax errors are created as the program is typed. No user action is necessary to trigger syntax checking. HAHA syntax is described in detail in section *Language reference*.

4.3 Type errors

Error markers related to typechecking are created and updated only when the source file is saved.

4.4 VCGen errors

Error markers are also produced whenever the solver is unable to prove validity of a formula generated from the program. These markers are updated whenever the verification conditions generator is run. Marker description contains context information, which describes why the formula was generated. Example of such information can be seen below:

```
Verification condition is not valid
  Program correctness
    Correctness of bsearch
      Block at lines 12 - 37
        Loop statement at line 17
          Single iteration preserves invariants.
            Block at lines 21 - 34
              If statement at line 24
                Case 1 - condition holds.
                  Postconditions are valid.
                    still_ordered at line 19
```

This description tells us that the solver was unable to prove that a loop invariant named `still_ordered` holds if the first branch of a conditional statement in line `24` is taken.

Verification conditions generator produces error markers in two cases

- When the solver was able to prove the invalidity of a formula. In this case a counterexample is produced and appended to error description. A counterexample is simply an assignment of values to free varialbes occuring in an expression. For example, an attempt to validate the following program

  ```
  function test(x : Z) : Z
    postcondition test >= x
      test := 3 * x
  ```

  would result in this error:

  ```
  Verification condition is not valid
  Program correctness
    Correctness of test
      Postcondition at line 2 (after substitution)

  Counterexample:
    x = (- 1)
  ```

- When the solver could neither prove nor disprove the formula. This case also covers problems caused by timeouts, internal solver errors and bugs in HAHA.

In case the program under consideration contains arrays, the counterexamples contain information about arrays. A statement of the form:

```
A = (_ as-array k!17),
```

means that the local variable `A` that holds an array is represented in the further lines of the counterexample as a function under the variable `k!17`. There is no special meaning hidden neither in the name `k` nor in the number `17`. These are random values from the point of view of the verification process.

Subsequently, statements that describe the variable `k!17` follow. Typically, we can observe here a formula like this:

```
k!17 = 0,
```

and this means that all the cells of the array are equal to 0. Another typical statement is:

```
k!17(x!1) = (ite (= x!1 0) 7 1)
```

This statement means that the array `k!17` is actually a function that assumes 7 when applied to 0 and assumes 1 in any other situation. Similarly:

```
k!17(x!1) = (ite (>= x!1 0) 5 2)
```

states that the function assumes 5 for all non-negative arguments and 2 for negative ones (yes, as the arrays are indexed with `Z`, the indices can be negative numbers). Of course, the `ite` expressions can be combined as in:

```
k!17(x!1) = (ite (>= x!1 0) (ite (>= x!1 2) 5 3) 2)
```

which represents the array that has 2 on negative indices, 3 on indices 0, 1, and 5 on positive indices starting with 2. It should not come as surprise that the assertion:

```
k!17(x!1) = (k!17!19 (k!18 x!1))
```

says that `k!17` is a composition of the arrays `k!18` and `k!17!19` understood as a composition of functions.

4.5 Console output

When the verification condition generator is run, a console is displayed in the lower pane. This console logs all messages printed during the process of creating and discharging verification conditions. Messages in the console are more detailed (but somewhat harder to read) than error marker descriptions.

First, all computed verification conditions are printed. Each condition consists of a list of assumptions, followed by a goal. Goal is separated from assumptions by a horizontal line. Note that all expressions are displayed in prefix notation. The following example shows a single verification condition:

```
n_ge_one : (>= len 1)
orderedA : (CALL-PREDICATE ordered A 1 len)
is_inA : (CALL-PREDICATE is_in A v 1 len)
---------------------------------------------
(AND (= 1 1) (AND (>= len 1) (AND (CALL-PREDICATE ordered A 1 len) (CALL-PREDICATE is_in A v 1 le
```

Declared variables, axioms and predicates are also printed at this stage.

In the second stage, computed conditions are discharged by the solver. During this stage, commands sent to the solver are printed. Possible commands include

- Declaration of a variable or predicate (`DECLARE VAR`)
- New assumption (`DECLARE AXIOM`)
- Context management commands `MARK` and `RESET`. The latter removes all declarations (both variables and assumptions) since last unmatched occurence of the former.
- Validity check `IS_VALID`. This command checks if the goal formula is valid in the current context. After this command, solver response (including a counterexample, if applicable) is printed to the console.

Text lines preceded by >> represent raw text sent to the external SMT solver. Similarly, lines starting with << contain raw solver responses. This is mainly useful for finding bugs related to translation of solver commands to the SMT2 format.

CHAPTER

FIVE

LANGUAGE REFERENCE

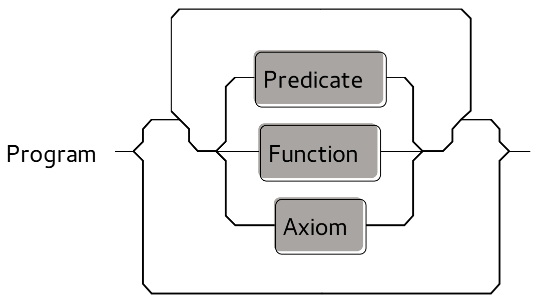
This section contains a complete description of HAHA syntax. All language constructs are presented using syntax diagrams.

5.1 Lexical conventions

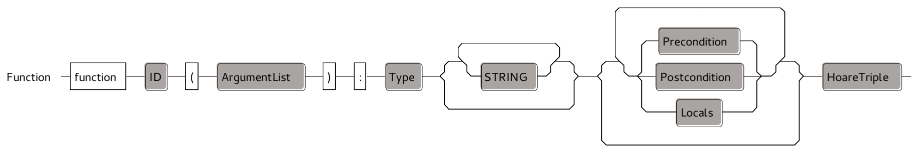
The following list specifies base lexical elements from which HAHA programs are composed.

- Whitespace, which is only used to separate other elements.
- Comments - both single line (`// ...`) and multiline (`/* ... */`). Multiline comments cannot be nested.
- Keywords and operators (specified in single quotes in the following text).
- Identifiers (`ID`), which consist of letters (a to z), digits and underscores. Identifiers cannot start with a digit.
- Numbers (BIGINT) - sequences of digits, leading zeros are not allowed.
- Strings (STRING) - which are used for documentation purposes only (there is no string type in the language). Strings are specified in qither double or single quotes, both variants support a set of escape sequences (e.g. “”” is a string consisting of a quote character).

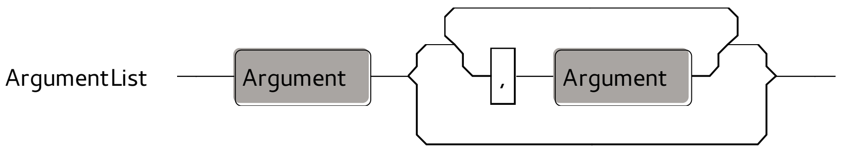
5.2 Program structure

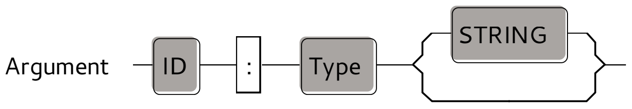


Each HAHA source file contains a single program, which is a list of top-level elements. A top-level element can be a function, a predicate or an axiom.



A function definition consists of





- Name.
- List of argument names and types.
- Result type.
- Optional documentation strings.
- Preconditions and postconditions
- Local variable declarations (each variable has name, type and an optional docstring).
- Function body.

A predicate is a named and parameterized boolean expression. Use of predicates can be very helpful in making a program readable. This is especially true when dealing with arrays, as in examples *Binary search* and *Quicksort partition*.

Axioms are simply boolean formulae that can be optionally named for clarity. Axioms are typically used when the automated solver is unable to prove a valid formula. Examples that make use of axioms include *Cubic root* and *Exponentiation*.

5.3 Types

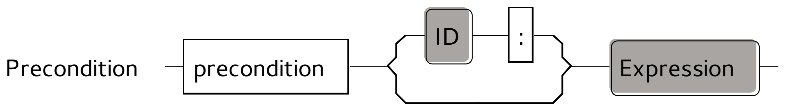
Basic types used in HAHA programs are true integers `Z` and multi-dimensional arrays. Booleans are also supported. The syntax also includes a primitive type `Int`, which is supposed to represent 32 bit integers, but support for that type is incomplete in current version of HAHA.

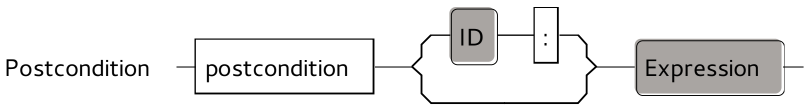
5.4 Hoare triples

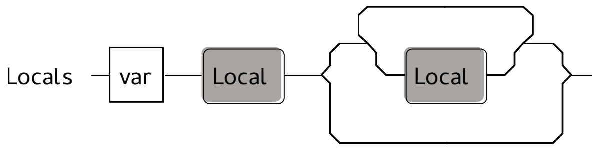
A Haore triple is a statement with a list of precondtions and postconditions. One peculiarity of the HAHA syntax is that a semicolon is necessary to terminate the list of preconditions, unless that list happens to be empty.

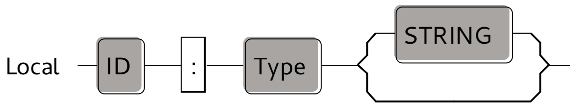
Statements available in HAHA include

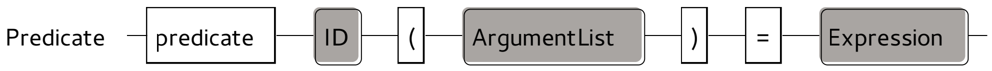
- The `skip` statement.
- Assignment - this includes assignment of the function result, which is represented by a special variable with the same name as the containing function.
- Blocks - lists of statements separated by midconditions.
- Conditional statements, with an optional `else` part.
- While loops. Loops are annotated with (optionally named) invariants and counter formulae. Counter formulae are used to prove termination of a program. This is an experimental feature and is not described in this documentation.

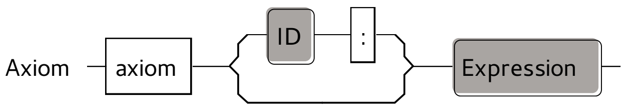


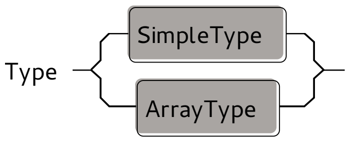

Postcondition
postcondition
ID
:
Expression

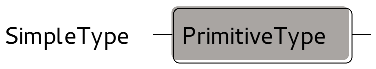

Locals
var
Local
Local

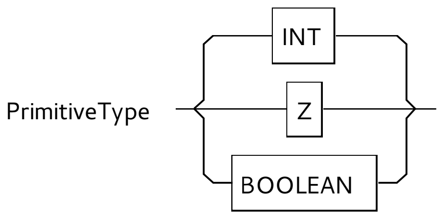

Local
ID
:
Type
STRING

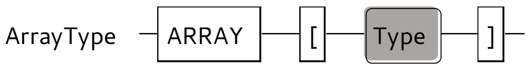

Predicate
predicate
ID
(
ArgumentList
)
=
Expression

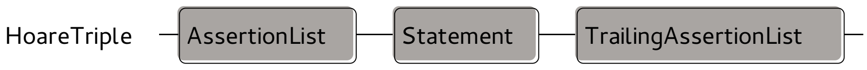

Axiom
axiom
ID
:
Expression

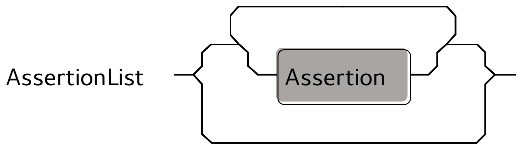

Type
SimpleType
ArrayType

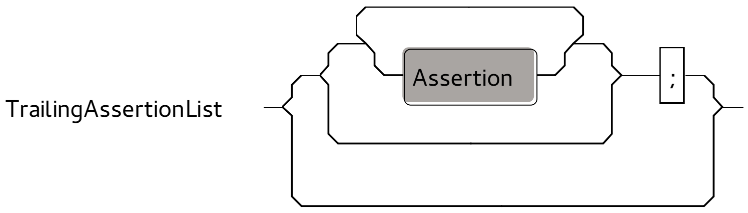

SimpleType
PrimitiveType

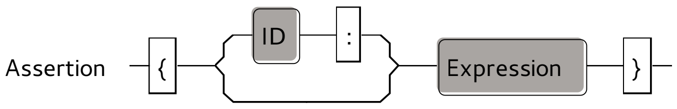

PrimitiveType
INT
Z
BOOLEAN

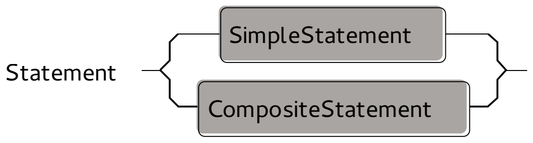

ArrayType
ARRAY
[
Type
]

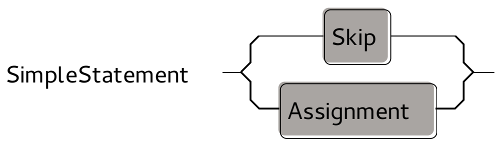

HoareTriple
AssertionList
Statement
TrailingAssertionList

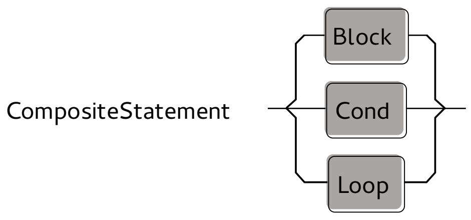

AssertionList
Assertion

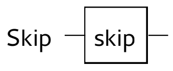

TrailingAssertionList
Assertion
;

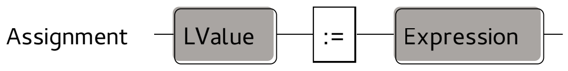

Assertion
{
ID
:
Expression
}

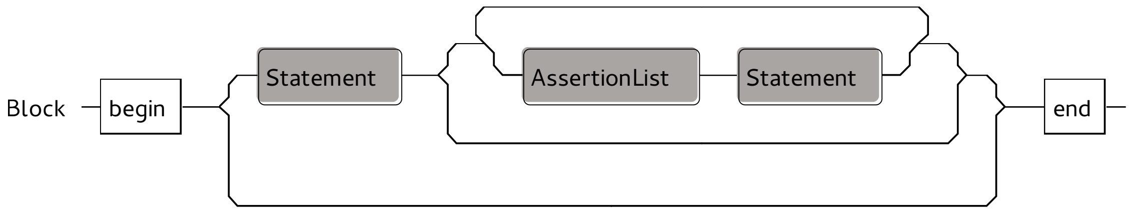

Statement
SimpleStatement
CompositeStatement

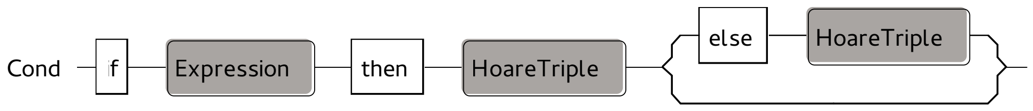

SimpleStatement
Skip
Assignment

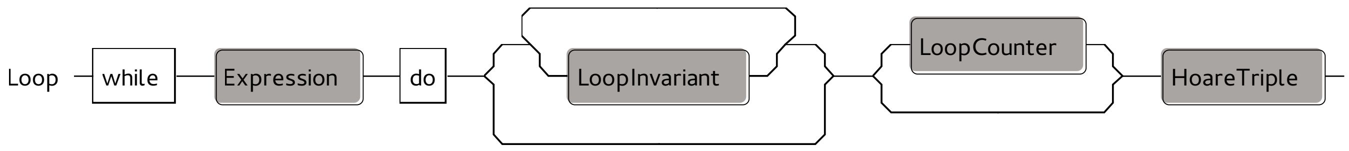

CompositeStatement
Block
Cond
Loop

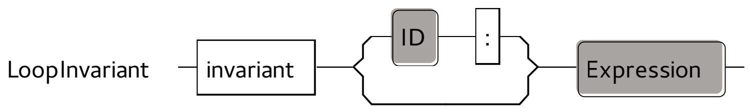

Skip
skip


Assignment
LValue
:=
Expression


Block
begin
Statement
AssertionList
Statement
end

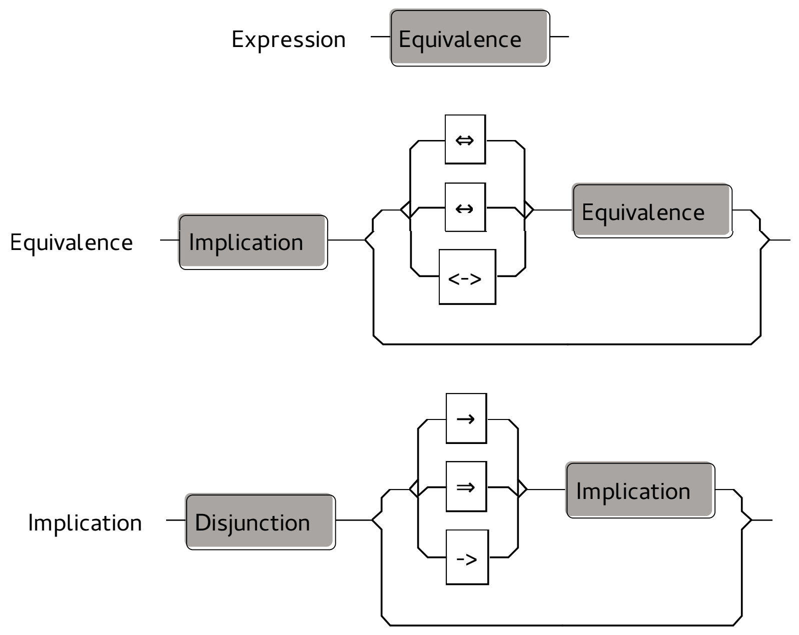

Cond
if
Expression
then
HoareTriple
else
HoareTriple


Loop
while
Expression
do
LoopInvariant
LoopCounter
HoareTriple


LoopInvariant
invariant
ID
:
Expression

There are some ambiguities in the concrete syntax corresponding to the diagrams presented hare. First, there is the dangling `else` problem, which is resolved by matching every `else` to nearest open `if`. Second issue arises when a statement, such as `if` or `while`, ends in a Hoare triple. If hat statement is itself a part of a Hoare triple, it might not be clear if trailing assertions should become a part of the inner or the outer triple. By default, the inner triple is chosen.

5.5 Expressions



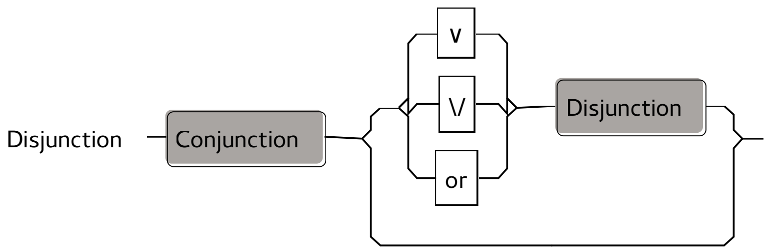
HAHA supports a number of traditional boolean and integer operators. It should be noted that many operators can be written in multiple variants. For example, conjunction can be represented as ∧ (Unicode character), `/\` (Coq style) or simply `and`.

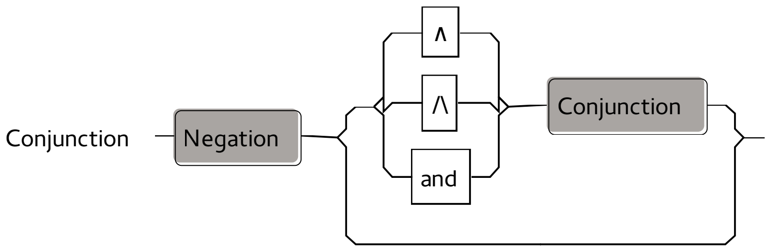
Supported operators include

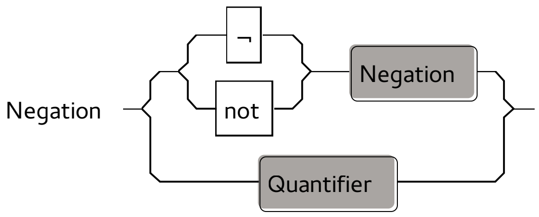
- Logical operators: iff, implication, conjunction, disjunction, negation.
- Quantifiers (`forall` and `exists`).
- Integer comparisons.
- Integer operations: addition, subtraction, multiplication, division, remainder and exponentiation.

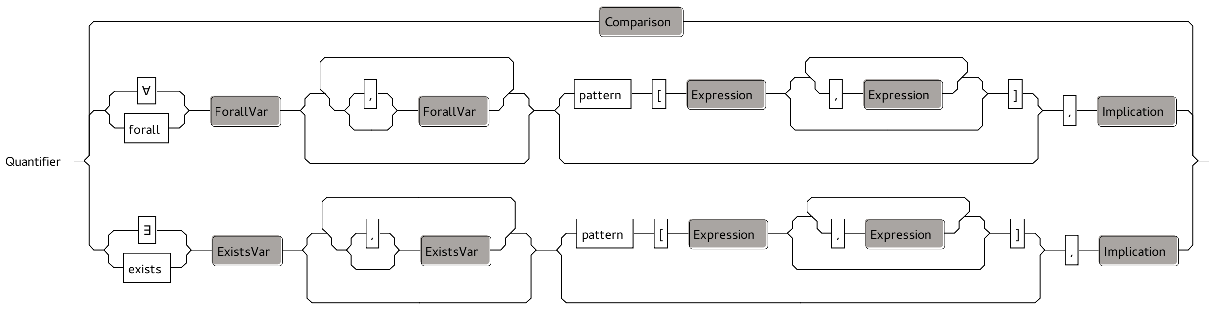
Quantified formulae may contain declarations of instantiation patterns. These patterns are used by the solver to decide when to create instances of a quantified formula. A pattern should contain all variables bound by the quantifier. For further discussion of the instantiation pattenrs, refer to the documentation of Simplify or Z3 solvers.

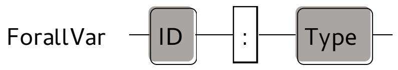
Note: Function calls are not supported by the current version of HAHA.

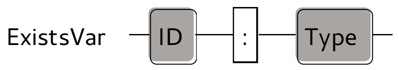

Disjunction
Conjunction
∨
\/
or
Disjunction

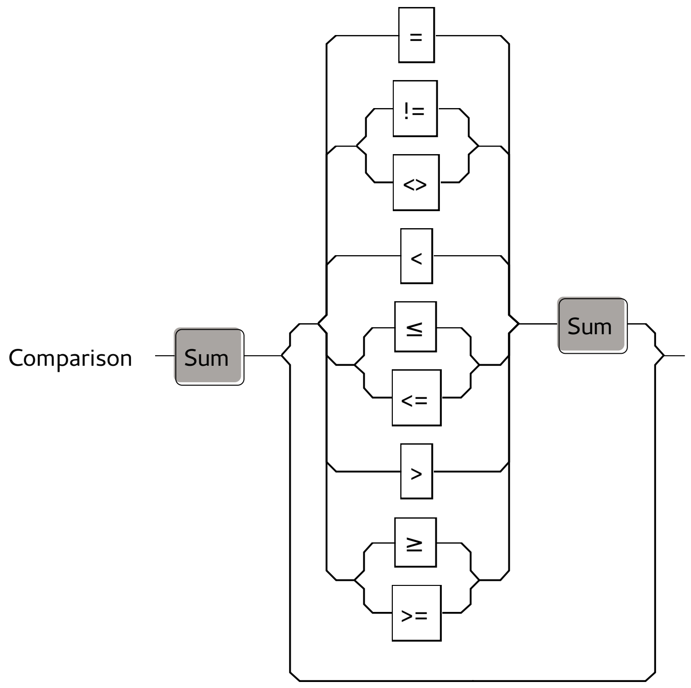

Conjunction
Negation
∧
/\
and
Conjunction

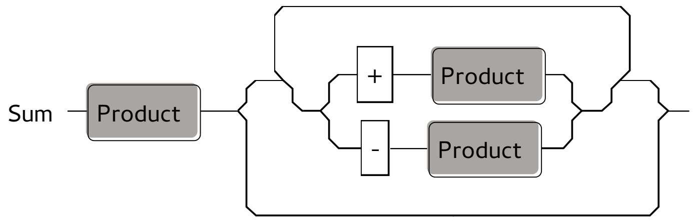

Negation
¬
not
Negation
Quantifier

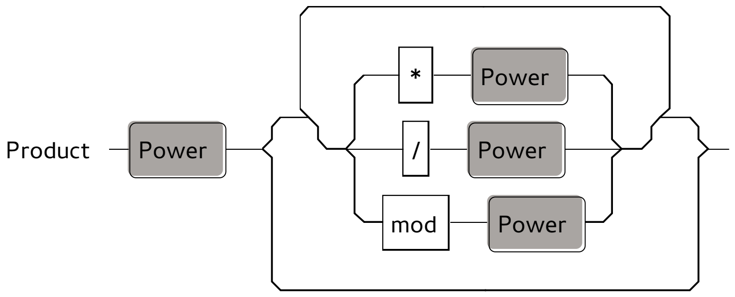

Quantifier
Comparison
∀
forall
ForallVar
,
ForallVar
pattern
[
Expression
,
Expression
]
,
Implication
∃
exists
ExistsVar
,
ExistsVar
pattern
[
Expression
,
Expression
]
,
Implication

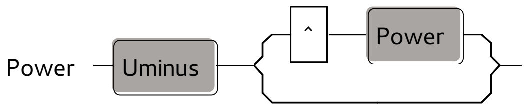

ForallVar
ID
:
Type

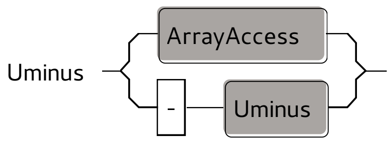

ExistsVar
ID
:
Type

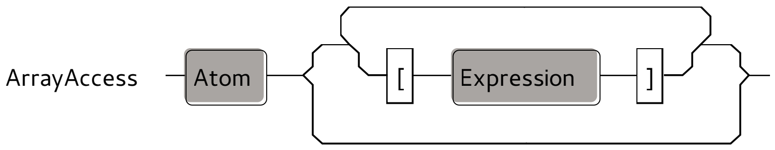

Comparison
Sum
=
!=
<>
<
≤
<=
>
≥
>=
Sum

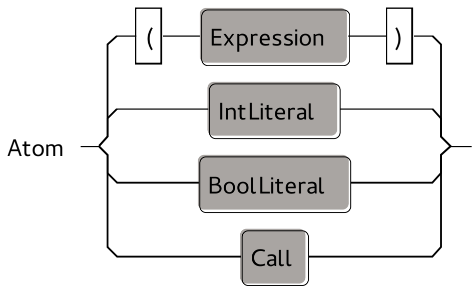

Sum
Product
+
Product
-
Product

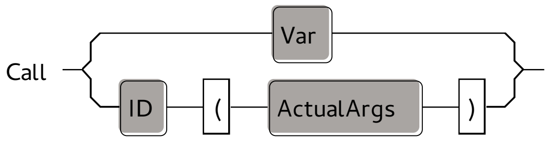

Product
Power
*
Power
/
Power
mod
Power

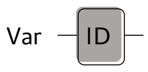

Power
Uminus
^
Power

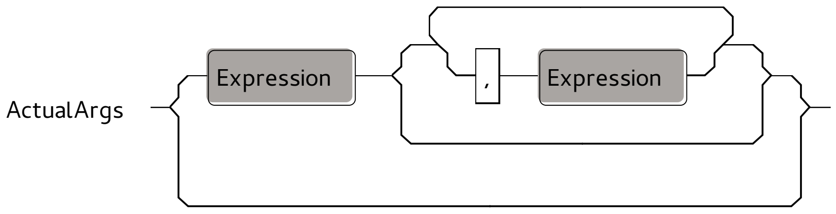

Uminus
ArrayAccess
-
Uminus

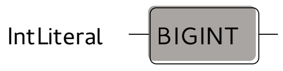

ArrayAccess
Atom
[
Expression
]

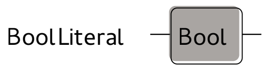

Atom
(
Expression
)
IntLiteral
BoolLiteral
Call

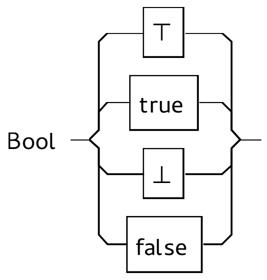

Call
Var
ID
(
ActualArgs
)


Var
ID


ActualArgs
Expression
,
Expression


IntLiteral
BIGINT


BoolLiteral
Bool


Bool
⊤
true
⊥
false

CHAPTER

SIX

EXAMPLES

Here is a number of examples that illustrate a few specific features of the HAHA language.

6.1 Cubic root

```
axiom cubicbin: forall b : Z, (b + 1) ^ 3 = b ^ 3 + 3 * (b ^ 2) + 3 * b + 1
axiom squarebin: forall b : Z, (b + 1) ^ 2 = b ^ 2 + 2 * b + 1

function croot( x : Z ) : Z
        precondition x >= 0
        postcondition (croot-1)^3 <=  x /\ x < croot^3
        var a : Z
                b : Z
                y : Z
begin
        a := 1
        { x >= 0 /\ a = 1 }
        b := 1
        { x >= 0 /\ a = 1 /\ b = 1 }
        y := 1
        { x >= 0 /\ a = 1 /\ b = 1 /\ y = 1}
        while y <= x do
        invariant (b-1)^3 <=  x /\  y = b^3 /\ a = b ^ 2 /\ x >= 0
        begin
                  y := y + 3*a + 3*b + 1 // y := (b + 1) ^3
                  { (b ^ 3) <=  x /\ y = (b + 1) ^ 3 /\
                    a  = b ^ 2  /\ x >= 0 }
                  a := a + 2*b + 1
                  { (b ^ 3) <=  x /\ y = (b +1) ^ 3 /\
                    a = (b+1)^2 /\ x >= 0 }
                  b := b + 1
          end
        { (b-1) ^ 3 <=  x /\ x < b ^3 }
        croot := b
end
```

The Z3 solver that is used as the proving backend for the tool supports well arithmetic with frequent addition. Therefore, we need to add additional information that helps the solver to complete the Hoare logic proofs in this example. This is done with axioms. They are introduced with `axiom` keyword and can be named as it is in the case of preconditio, postcondition or axiom.

The assertions can span over many lines, which can be observed in the loop body of the example:

```
y := y + 3*a + 3*b + 1 // y := (b + 1) ^3
{ (b ^ 3) <=  x /\ y = (b + 1) ^ 3 /\
  a  = b ^ 2  /\ x >= 0 }
a := a + 2*b + 1
```

We can also supply additional comments that provide additional explanation for particular assertions or pieces of code:

```
y := y + 3*a + 3*b + 1 // y := (b + 1) ^3
```

Here, we inform that the assignment is in fact the assignment of the cubic power of `b + 1` to `y`.

6.2 Exponentiation

```
predicate odd(x : Z) = (x mod 2 = 1)

axiom tozero: forall z : Z p : Z, z * (p ^ 0) = z
axiom odddiv2: forall z : Z p : Z q : Z,
  odd(q) -> z * p ^ q = z * p ^ (2 * (q / 2) + 1)
axiom twoinexp: forall z : Z p : Z q : Z, z * p ^ (2 * q) = z * (p * p ) ^ q
axiom twoandoneinexp: forall z : Z p : Z q : Z,
  z * (p ^ ((2 * q) + 1)) = z * ((p * p) ^ q) * p

function power( y : Z, x : Z ) : Z
  precondition n_ge_one : y >= 0
  precondition xnz : x <> 0
  postcondition power = x ^ y
  var z : Z
      p : Z
      q : Z
begin
  z := 1
  { p = p /\ z = 1 /\ y >= 0 /\ x <> 0 }
  p := x
  { q = q /\ p = x /\ z = 1 /\ y >= 0 /\ p <> 0 }
  q := y
  { q = y /\ p = x /\ z = 1 /\ y >= 0 /\ p <> 0 }
  while q <> 0 do
  invariant z * p ^ q = x ^ y /\ q >= 0 /\ y >= 0 /\ p <> 0
  begin
      if q mod 2 = 1 then
          begin
        z := z * p
      end
      { (odd(q) /\ q = 2 * (q / 2) + 1 /\ (z * p ^ q = (x ^ y) * p ))
        \/
        (not odd(q) /\ (z * p ^ (2 * (q / 2)) = x ^ y)) }
      { q > 0 }
      { y >= 0 }
      { p <> 0 }
      q := q / 2
      { z * (p * p) ^ q = x ^ y }
      { q >= 0 }
      { y >= 0 }
      { p <> 0 }
      p := p * p
    end
    { z * (p ^ q) = x ^ y /\ q = 0 /\ y >= 0 }
    power := z
end
```

This example shows that we can give function multiple parameters. Unlike Pascal we require programmers to give full type specifications in case multiple parameters of the same type are necessary. Therefore, the header of the exponent function is:

```
function power( y : Z, x : Z ) : Z
```

Another important feature of the program is that we use here a simple predicate. The predicate is introduced with the keyword `predicate` and its header is similar to the one of function except that the result type is not given as it does not make sense here. The header of the predicate definition is followed by = sign and the definition of the predicate body. In our case this is:

```
predicate odd(x : Z) = (x mod 2 = 1)
```

We can see here a number of axioms that are necessary to make the proof go through.

- The axiom `tozero` makes it possible to draw the final conclusion from the exit condition of the while loop. The condition holds when `q` is zero and then the invariant condition gives the required result provided that we know the meaning of exponentiation to the power of zero.
- The axiom `odddiv2` is necessary to understand that odd powers decompose in a particular way when divided by 2. This formula is necessary to make proof go through the instruction `q := q / 2`.
- The axioms `twoinexp` and `twoandoneinexp` are necessary to understand how even and odd powers turn to multiplication. This is again necessary for the proof of the instruction `q := q / 2`.

6.3 Binary search

```
predicate ordered(A : ARRAY [Z], i : Z, j : Z) = forall x : Z y : Z,
    i <= x -> x <= y -> y <= j -> A[x] <= A[y]
predicate is_in(A : ARRAY [Z], v : Z, i : Z, j : Z) = exists x : Z,
    (i <= x /\ x <= j) /\ A[x] = v

function bsearch( A : ARRAY [Z], len : Z , v : Z ) : Z
  precondition n_ge_one : len >= 1 //lenght ia st least 1
  precondition orderedA: ordered(A, 1, len)//array is ordered
  precondition is_inA: is_in(A, v, 1, len) //the value we look for is in the array
  postcondition A[bsearch] = v //the result is a pointer to the value we look for
  var i : Z
      j : Z
      k : Z
begin
  i := 1
  { i = 1 /\ len >= 1 /\ ordered(A, 1, len) /\ is_in(A, v, 1, len) }
  j := len
  { i = 1 /\ j = len /\ len >= 1 /\ ordered(A, 1, len) /\ is_in(A, v, 1, len) }
  while i < j do
    invariant is_in(A, v, i, j) /\ ordered(A, i, j) /\ i<=j
  begin
      k := (i + j) / 2
      { is_in(A, v, i, j) /\ ordered(A, i, j) /\ i<j /\ k = (i+j)/2 }
      if A[k] < v then
        i := k + 1
      else
        j := k
    end
    { is_in(A, v, i, i) }
    bsearch := i
end
```

This example illustrates the way we handle arrays. The array parameter declaration in predicates and functions looks as follows:

```
predicate ordered( A : ARRAY [Z], i : Z, j : Z ) = ...
function bsearch( A : ARRAY [Z], len : Z , v : Z ) : Z
```

One important thing to note here is that the arrays in our language have infinite domain of keys, namely the whole integer numbers `Z`. Therefore, we need to introduce explicit parameter that describes the range of the array to make the verification similar to the real-world situation. In this example we decided that the arrays range from `1` to some number held in the parameter `len`.

This example also illustrates how well chosen set of predicates may make the verified procedure very nicely and comprehensively documented.

6.4 Quicksort partition

```
predicate left_contains_le (A : ARRAY[Z], where : Z, val : Z) =
  forall i : Z , (1 <= i /\ i < where) -> A[i] <= val

predicate right_contains_gt (A : ARRAY[Z], where : Z, bound : Z, val : Z) =
  forall i : Z, (where < i /\ i <= bound) -> A[i] > val

predicate sides_parted(A : ARRAY[Z], left : Z, right :Z, bound : Z, val : Z) =
  left_contains_le(A, left, val) /\ right_contains_gt(A, right, bound, val)

predicate is_copy(A : ARRAY[Z], B : ARRAY[Z], len : Z) =
  forall i : Z, (1 <= i /\ i <= len) -> A[i] = B[i]

predicate between(l : Z, m : Z, r : Z) = l <= m /\ m <= r


function partition(A : ARRAY[Z], len : Z) : ARRAY[Z]
  precondition  1 <= len
  postcondition
    exists k : Z, sides_parted(partition, k, k, len, A[1])
  var v : Z
      i : Z
      j : Z
      x : Z
begin
  partition := A
  { is_copy(A, partition, len) /\ 1 <= len }
  v := A[1]
  { is_copy(A, partition, len) /\ 1 <= len /\ v = partition[1] /\ v = A[1] }
  partition[len+1] := v + 1
  { is_copy(A, partition, len) /\ 1 <= len /\ v = partition[1] /\ v = A[1] /\
    partition[len+1] > v
  }
  i := 2
  { is_copy(A, partition, len) /\ 1 <= len /\ v = partition[1] /\ v = A[1] /\
    partition[len+1] > v /\ i = 2
  }
  j := len
  { is_copy(A, partition, len) /\ 1 <= len /\ v = partition[1] /\ v = A[1] /\
    partition[len+1] > v /\ i = 2 /\ j = len
  }
  x := 0
  { is_copy(A, partition, len) /\ 1 <= len /\ v = partition[1] /\ v = A[1] /\
    sides_parted(partition, i, j, len, v) /\
    partition[len+1] > v /\ i = 2 /\ j = len /\ x = 0
  }
  while i <= j do
    invariant pointers: j + 1 >= i - 1 /\ between(2, i, len+1) /\ between(1, j, len)
    invariant sides: sides_parted(partition, i, j, len, v)
    invariant guards: partition[1] = v /\ partition[len+1] > v
    invariant general: 1 <= len /\ v = A[1]
  begin
```

```
    while partition[i] <= v do
      invariant pointers: j+1 >= i - 1 /\ between(2, i, len+1) /\ between(1, j, len)
      invariant sides: sides_parted(partition, i, j, len, v)
      invariant guards: partition[1] = v /\ partition[len+1] > v
      invariant general: 1 <= len /\ v = A[1]
    begin
      i := i + 1
    end
    { j+1 >= i - 1 /\ between(2, i, len+1) /\ between(1, j, len) /\
      sides_parted(partition, i, j, len, v) /\
      partition[1] = v /\ partition[len+1] > v /\ partition[i] > v /\
      1 <= len /\ v = A[1] }
    while partition[j] > v do
      invariant pointers: j + 1 >= i - 1 /\ between(2, i, len+1) /\ between(1, j, len)
      invariant sides: sides_parted(partition, i, j, len, v)
      invariant guards: partition[1] = v /\ partition[len+1] > v /\ partition[i] > v
      invariant general: 1 <= len /\ v = A[1]
    begin
      j := j - 1
    end
    { j + 1 >= i - 1 /\ between(2, i, len+1) /\ between(1, j, len) /\
      sides_parted(partition, i, j, len, v) /\
      partition[1] = v /\ partition[len+1] > v /\ partition[i] > v /\ partition[j] <= v /\
      1 <= len /\ v = A[1] }
    if i < j then
    begin
      x := partition[i]
      { 1 < i /\ i < j /\ between(2, i, len+1) /\ between(1, j, len) /\
        sides_parted(partition, i, j, len, v) /\
        partition[1] = v /\ partition[len+1] > v /\ partition[j] <= v /\
        x = partition[i] /\ x > v /\
        1 <= len /\ v = A[1] }
      partition[i] := partition[j]
      { 1 < i /\ i < j /\ between(2, i, len+1) /\ between(1, j, len) /\
        sides_parted(partition, i + 1, j, len, v) /\
        partition[1] = v /\ partition[len+1] > v /\ partition[j] <= v /\
        x > v /\
        1 <= len /\ v = A[1] }
      partition[j] := x
      { 1 < i /\ i < j /\ between(2, i, len+1) /\ between(1, j, len) /\
        sides_parted(partition, i + 1, j - 1, len, v) /\
        partition[1] = v /\ partition[len+1] > v /\
        x > v /\ 1 <= len /\ v = A[1] }
      i := i + 1
      { 1 < i /\ i < j + 1 /\ between(2, i, len+1) /\ between(1, j, len) /\
        sides_parted(partition, i, j - 1, len, v) /\
        partition[1] = v /\ partition[len+1] > v /\
        x > v /\
        1 <= len /\ v = A[1] }
      j := j - 1
    end
  end
  { i > j /\ j + 1 >= i - 1 /\ between(2, i, len+1) /\ between(1, j, len) /\
    sides_parted(partition, i, j, len, v) /\
    partition[1] = v /\ partition[len+1] > v /\
    1 <= len  /\ v = A[1] }
  partition[1] := partition[j]
  { i > j /\ j + 1 >= i - 1 /\ between(2, i, len+1) /\ between(1, j, len) /\
    sides_parted(partition, i, j, len, v) /\
    partition[len+1] > v /\
    1 <= len  /\ v = A[1] }
  partition[j] := v
end
```

This example serves to demonstrate how the features such as predicates and invariant labelling work in bigger examples. First, the predicates make it possible to exchange long and obscure expressions that define particular features into insightful labels. Second, the complicated invariant formula can be divided into meaningful pieces that describe a particular aspects of the loop invariant, which also contributes to readability. It is important to understand that the split does not mean that the reasoning about the pieces is separate. In particular we need the information on guards to correctly establish the new version of `sides` part.

We have to make one important point. It is critical to split the invariants when they become so big since it is very difficult without this to figure out what is wrong in case some subtle mistake is done either in the code of the program or in the assertion formula. Splitting of the conditions helps to localise the problem.