The Why3 platform

Version 1.2.1, October 2019

François Bobot[1,2]
Jean-Christophe Filliâtre[1,2]
Claude Marché[2,1]
Guillaume Melquiond[2,1]
Andrei Paskevich[1,2]

[1] LRI, CNRS & University Paris-Sud, Orsay, F-91405
[2] Inria Saclay – Île-de-France, Palaiseau, F-91120

This work has been partly supported by the 'U3CAT' national ANR project (ANR-08-SEGI-021-08, http://frama-c.com/u3cat/), the 'Hi-Lite' (http://www.open-do.org/projects/hi-lite/) FUI project of the System@tic competitivity cluster, the 'BWare' ANR project (ANR-12-INSE-0010, http://bware.lri.fr/), the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse); the 'CoLiS' ANR project (ANR-15-CE25-0001, http://colis.irif.univ-paris-diderot.fr/), and the 'VOCaL' ANR project (ANR-15-CE25-008, https://vocal.lri.fr/).

Foreword

Why3 is a platform for deductive program verification. It provides a rich language for specification and programming, called WhyML, and relies on external theorem provers, both automated and interactive, to discharge verification conditions. Why3 comes with a standard library of logical theories (integer and real arithmetic, Boolean operations, sets and maps, etc.) and basic programming data structures (arrays, queues, hash tables, etc.). A user can write WhyML programs directly and get correct-by-construction OCaml programs through an automated extraction mechanism. WhyML is also used as an intermediate language for the verification of C, Java, or Ada programs.

Why3 is a complete reimplementation of the former Why platform [6]. Among the new features are: numerous extensions to the input language, a new architecture for calling external provers, and a well-designed API, allowing to use Why3 as a software library. An important emphasis is put on modularity and genericity, giving the end user a possibility to easily reuse Why3 formalizations or to add support for a new external prover if wanted.

Availability

Why3 project page is `http://why3.lri.fr/`. The last distribution is available there, in source format, together with this documentation and several examples.

Why3 is also distributed under the form of an OPAM package and a Debian package.

Why3 is distributed as open source and freely available under the terms of the GNU LGPL 2.1. See the file `LICENSE`.

See the file `INSTALL` for quick installation instructions, and Section 4 of this document for more detailed instructions.

Contact

There is a public mailing list for users' discussions: `http://lists.gforge.inria.fr/mailman/listinfo/why3-club`.

Report any bug to the Why3 Bug Tracking System: `https://gitlab.inria.fr/why3/why3/issues`.

Acknowledgements

We gratefully thank the people who contributed to Why3, directly or indirectly: Stefan Berghofer, Sylvie Boldo, Martin Clochard, Simon Cruanes, Sylvain Dailler, Clément Fumex, Léon Gondelman, David Hauzar, Daisuke Ishii, Johannes Kanig, Mikhail Mandrykin, David Mentré, Benjamin Monate, Kim Nguyen, Thi-Minh-Tuyen Nguyen, Mário Pereira, Raphaël Rieu-Helft, Simão Melo de Sousa, Asma Tafat, Piotr Trojanek, Makarius Wenzel.

Contents

Part I

Tutorial

Chapter 1

Getting Started

1.1 Hello Proofs

The first step in using Why3 is to write a suitable input file. When one wants to learn a programming language, one starts by writing a basic program. Here is our first Why3 file, which is the file `examples/logic/hello_proof.why` of the distribution. It contains a small set of goals.

```
theory HelloProof

  goal G1: true

  goal G2: (true -> false) /\ (true \/ false)

  use int.Int

  goal G3: forall x:int. x * x >= 0

end
```

Any declaration must occur inside a theory, which is in that example called `HelloProof`. It contains three goals named G_1, G_2, G_3. The first two are basic propositional goals, whereas the third involves some integer arithmetic, and thus it requires to import the theory of integer arithmetic from the Why3 standard library, which is done by the `use` declaration above.

We don't give more details here about the syntax and refer to Chapter 2 for detailed explanations. In the following, we show how this file is handled in the Why3 GUI (Section 1.2) then in batch mode using the `why3` executable (Section 1.3).

1.2 Getting Started with the GUI

The graphical interface allows to browse into a file or a set of files, and check the validity of goals with external provers, in a friendly way. This section presents the basic use of this GUI. Please refer to Section 5.3 for a more complete description.

The GUI is launched on the file above as follows (here "`>`" is the prompt):

```
> why3 ide hello_proof.why
```

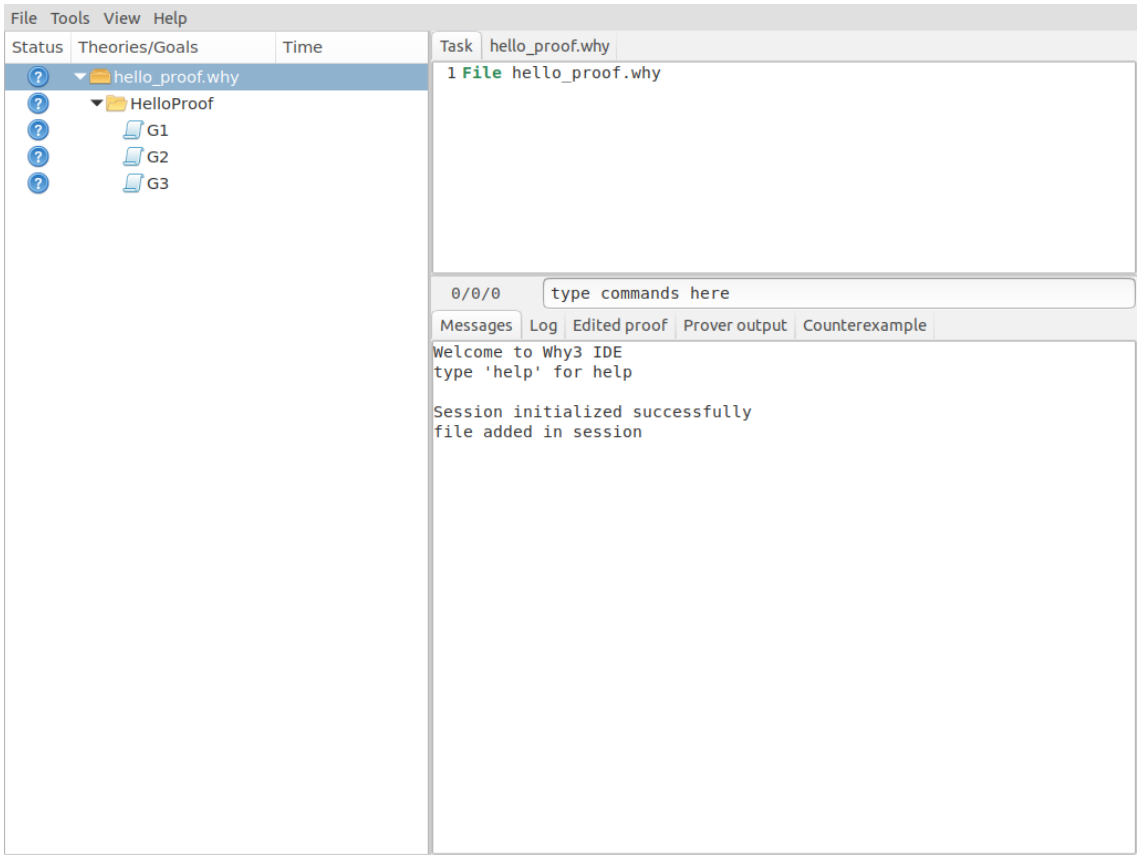

Figure 1.1: The GUI when started the very first time.

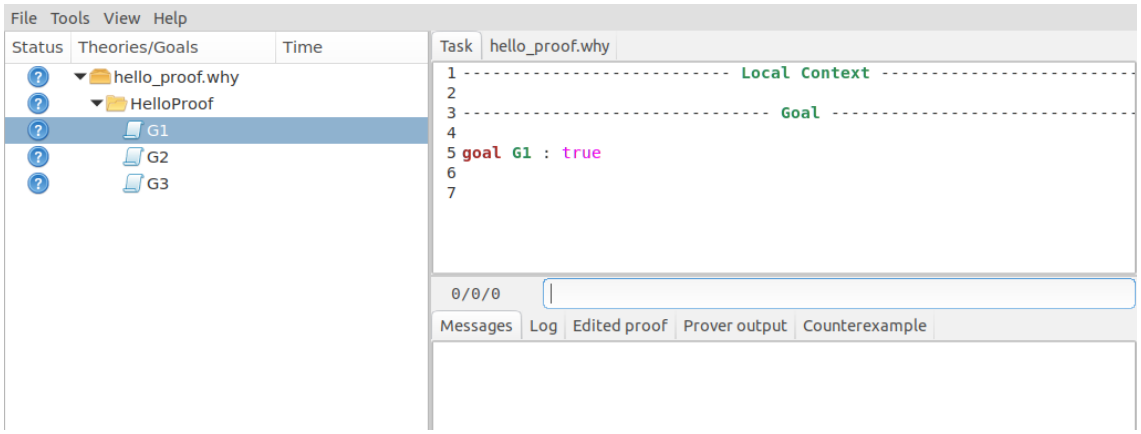

Figure 1.2: The GUI with goal G1 selected.

When the GUI is started for the first time, you should get a window that looks like the screenshot of Figure 1.1. The left part is a tree view that allows to browse inside the theories. In this tree view, we have a structured view of the file: this file contains one theory, itself containing three goals. In Figure 1.2, we clicked on the row corresponding to goal G_1. The *task* associated with this goal is then displayed on the top-right pane. The corresponding part of the input file is shown when clicking the rightmost tab of that pane.

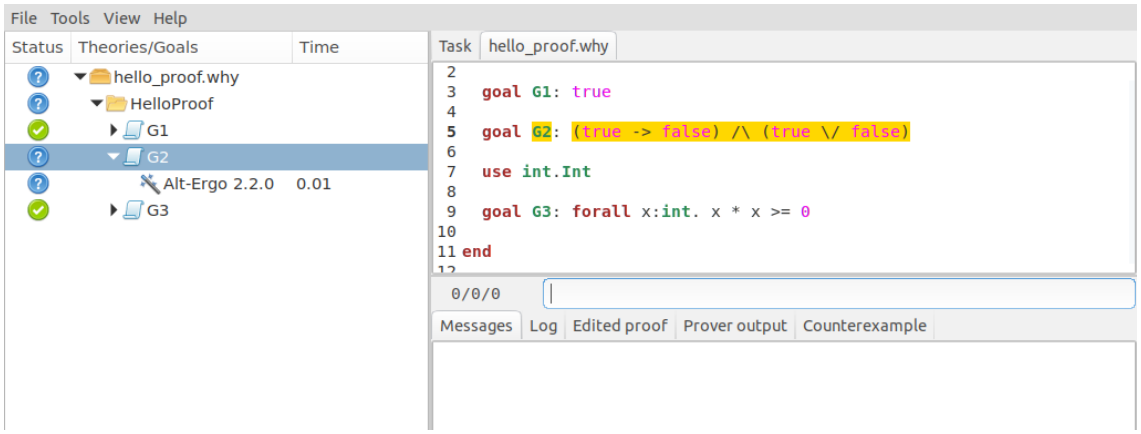

Figure 1.3: The GUI after running the Alt-Ergo prover on each goal.

1.2.1 Calling provers on goals

You are now ready to call provers on the goals [1]. A prover is selected using the context menu (right-click). This prover is then called on the goal selected in the tree view. You can select several goals at a time, either by using multi-selection (typically by clicking while pressing the Shift or Ctrl key) or by selecting the parent theory or the parent file.

Let us now select the theory "HelloProof" and run the Alt-Ergo prover. After a short time, you should get the display of Figure 1.3. Goals G_1 and G_3 are now marked with a green "checked" icon in the status column. This means that these goals have been proved by Alt-Ergo. On the contrary, goal G_2 is not proved; it remains marked with a question mark. You could attempt to prove G_2 using another prover, though it is obvious here it will not succeed.

1.2.2 Applying transformations

Instead of calling a prover on a goal, you can apply a transformation to it. Since G_2 is a conjunction, a possibility is to split it into subgoals. You can do that by selecting Split VC in the context menu. Now you have two subgoals, and you can try again a prover on them, for example Alt-Ergo. We already have a lot of goals and proof attempts, so it is a good idea to close the sub-trees which are already proved: this can be done by the menu View/Collapse proved goals, or even better by its shortcut "Ctrl-C". You should see now what is displayed on Figure 1.4.

The first part of goal G_2 is still unproved. As a last resort, we can try to call the Coq proof assistant, by selecting it in the context menu. A new sub-row appear for Coq, and the Coq proof editor is launched. (It is `coqide` by default; see Section 5.3 for details on how to configure this). You get now a regular Coq file to fill in, as shown on Figure 1.5. Please be mindful of the comments of this file. They indicate where Why3 expects you to fill the blanks. Note that the comments themselves should not be removed, as they are needed to properly regenerate the file when the goal is changed. See Section 8.3 for more details.

Of course, in that particular case, the goal cannot be proved since it is not valid. The only thing to do is to fix the input file, as explained below.

[1] If not done yet, you must perform prover autodetection using `why3 config --detect-provers`

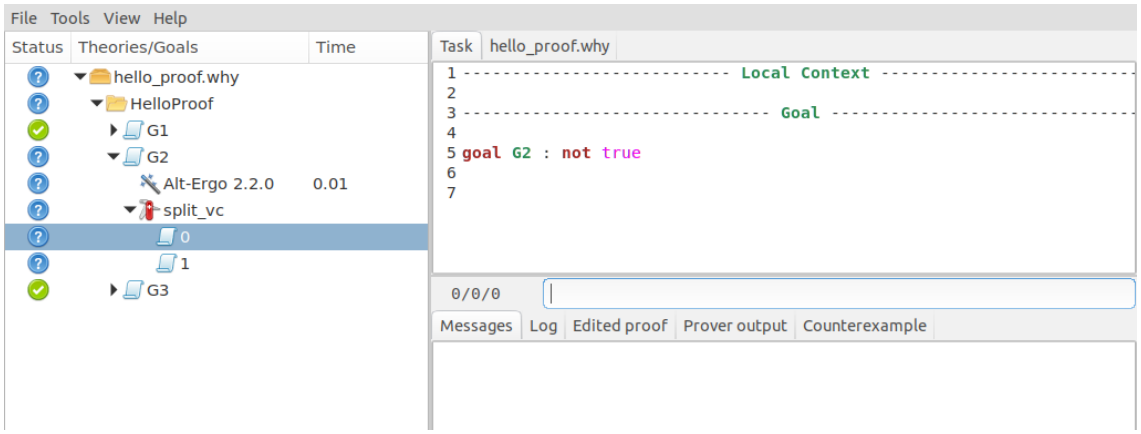

Figure 1.4: The GUI after splitting goal G_2.

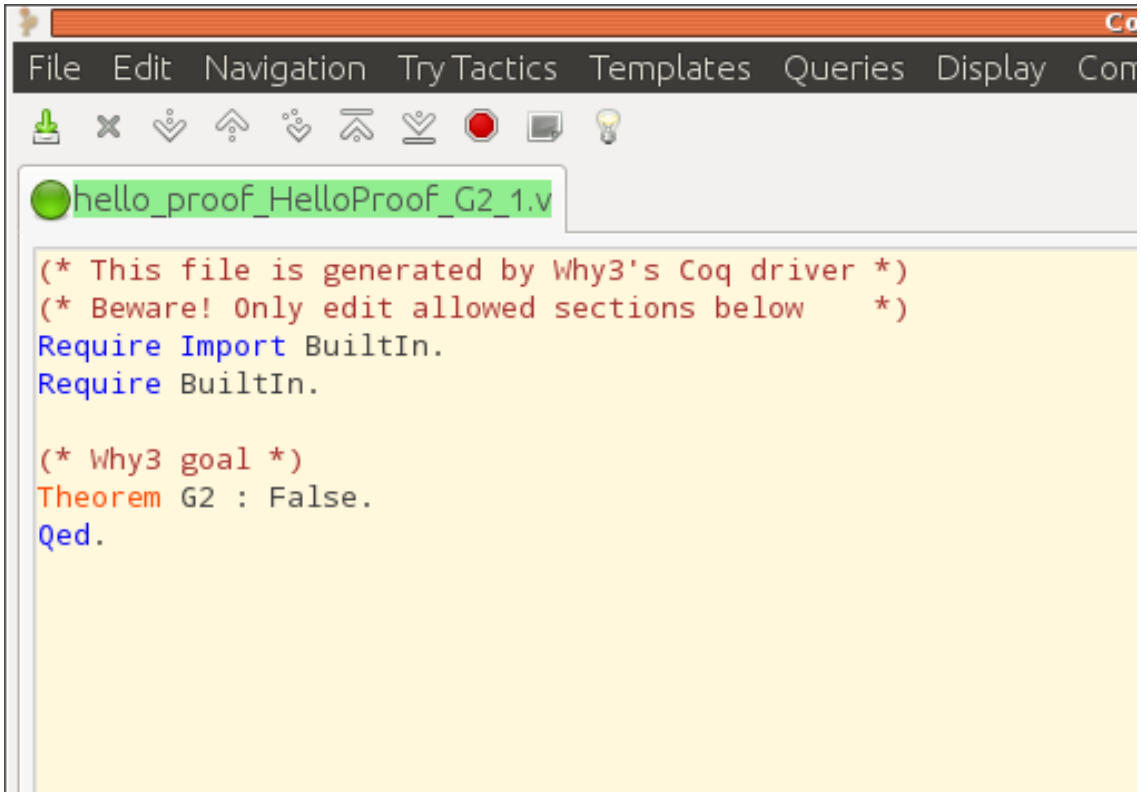

Figure 1.5: CoqIDE on subgoal 1 of G_2

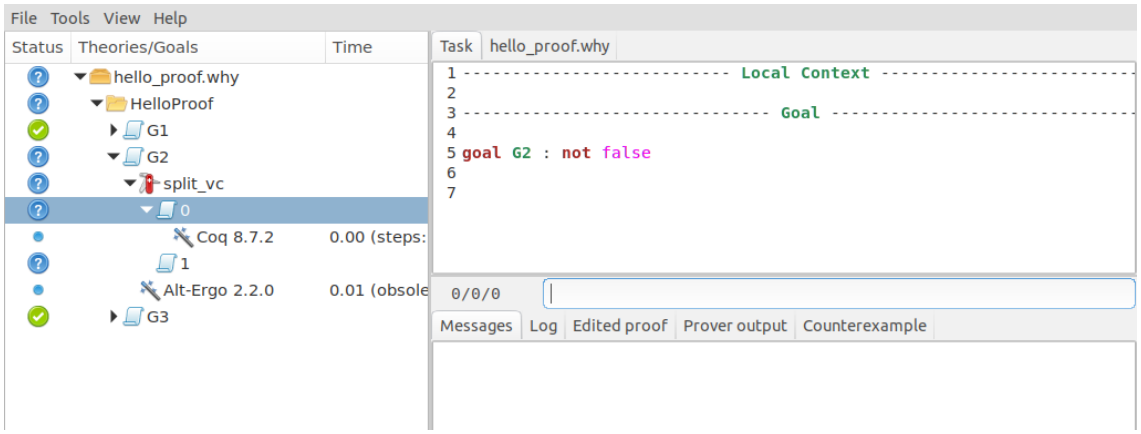

Figure 1.6: File reloaded after modifying goal G_2

1.2.3 Modifying the input

You can edit the source file, using the corresponding tab in the top-right window of the GUI. Let us assume we change the goal G_2 by replacing the first occurrence of `true` by `false`, *e.g.*

```
goal G2 : (false -> false) /\ (true \/ false)
```

We can refresh the goals using menu File/Save all and Refresh session, or the shortcut "Ctrl-R". We get the tree view shown on Figure 1.6.

The important feature to notice first is that all the previous proof attempts and transformations were saved in a database — an XML file created when the Why3 file was opened in the GUI for the first time. Then, for all the goals that remain unchanged, the previous proofs are shown again. For the parts that changed, the previous proofs attempts are shown but marked with "(obsolete)" so that you know the results are not accurate. You can now retry to prove all what remains unproved using any of the provers.

1.2.4 Replaying obsolete proofs

Instead of pushing a prover's button to rerun its proofs, you can *replay* the existing but obsolete proof attempts, using menu Tools/Replay obsolete. By default, Replay only replays proofs that were successful before.

Notice that replaying can be done in batch mode, using the `replay` command (see Section 5.4) For example, running the replayer on the `hello_proof` example is as follows (assuming G_2 still is `(true -> false) /\ (true \/ false)`).

```
> why3 replay hello_proof
 2/3 (replay OK)
   +--file ../hello_proof.why: 2/3
      +--theory HelloProof: 2/3
         +--goal G2 not proved
```

The last line tells us that no differences were detected between the current run and the run stored in the XML file. The tree above reminds us that G_2 is not proved.

1.2.5 Cleaning

You may want to clean some of the proof attempts, *e.g.* removing the unsuccessful ones when a project is finally fully proved. A proof or a transformation can be removed by selecting it and using menu Tools/Remove or the `Delete` key. Menu Tools/Clean or shortcut "C" perform an automatic removal of all proofs attempts that are unsuccessful, while there exists a successful proof attempt for the same goal. Beware that there is no way to undo such a removal.

1.3 Getting Started with the Why3 Command

The `prove` command makes it possible to check the validity of goals with external provers, in batch mode. This section presents the basic use of this tool. Refer to Section 5.2 for a more complete description of this tool and all its command-line options.

The very first time you want to use Why3, you should proceed with autodetection of external provers. On the command line, this is done as follows:

```
> why3 config --detect
```

This prints some information messages on what detections are attempted. To know which provers have been successfully detected, you can do as follows.

```
> why3 --list-provers
Known provers:
  Alt-Ergo 1.30
  CVC4 1.5
  Coq 8.6
```

The first word of each line is a unique identifier for the associated prover. We thus have now the three provers Alt-Ergo [4], CVC4 [1], and Coq [2].

Let us assume that we want to run Alt-Ergo on the HelloProof example. The command to type and its output are as follows, where the `-P` option is followed by the unique prover identifier (as shown by `--list-provers` option).

```
> why3 prove -P Alt-Ergo hello_proof.why
hello_proof.why HelloProof G1: Valid (0.00s, 1 steps)
hello_proof.why HelloProof G2: Unknown (other) (0.01s)
hello_proof.why HelloProof G3: Valid (0.00s, 1 steps)
```

Unlike the Why3 GUI, the command-line tool does not save the proof attempts or applied transformations in a database.

We can also specify which goal or goals to prove. This is done by giving first a theory identifier, then goal identifier(s). Here is the way to call Alt-Ergo on goals G_2 and G_3.

```
> why3 prove -P Alt-Ergo hello_proof.why -T HelloProof -G G2 -G G3
hello_proof.why HelloProof G2 : Unknown: Unknown (0.01s)
hello_proof.why HelloProof G3 : Valid (0.01s)
```

Finally, a transformation to apply to goals before proving them can be specified. To know the unique identifier associated to a transformation, do as follows.

```
> why3 --list-transforms
Known non-splitting transformations:
  [...]

Known splitting transformations:
  [...]
  split_goal_right
```

Here is how you can split the goal G_2 before calling Simplify on the resulting subgoals.

```
> why3 prove -P Alt-Ergo hello_proof.why -a split_goal_right -T HelloProof -G G2
hello_proof.why HelloProof G2: Unknown (other) (0.01s)
hello_proof.why HelloProof G2: Valid (0.00s, 1 steps)
```

Section 9.5 gives the description of the various transformations available.

Chapter 2

The WhyML Language

This chapter describes the WhyML specification and programming language. A WhyML source file has suffix `.mlw`. It contains a list of modules. Each module contains a list of declarations. These includes

- Logical declarations:
  - types (abstract, record, or algebraic data types);
  - functions and predicates;
  - axioms, lemmas, and goals.
- Program data types. In a record type declaration, some fields can be declared `mutable` and/or `ghost`. Additionally, a record type can be declared `abstract` (its fields are only visible in ghost code / specification).
- Program declarations and definitions. Programs include many constructs with no counterpart in the logic:
  - mutable field assignment;
  - sequence;
  - loops;
  - exceptions;
  - local and anonymous functions;
  - ghost parameters and ghost code;
  - annotations: pre- and postconditions, assertions, loop invariants.

  A program may be non-terminating. (But termination can be proved if we wish.)

Command-line tools described in the previous chapter also apply to files containing programs. For instance

```
> why3 prove myfile.mlw
```

displays the verification conditions for programs contained in file `myfile.mlw`, and

```
> why3 prove -P alt-ergo myfile.mlw
```

runs the SMT solver Alt-Ergo on these verification conditions. All this can be performed within the GUI tool `why3 ide` as well. See Chapter 5 for more details regarding command lines.

As an introduction to WhyML, we use a small logical puzzle (Section 2.1) and then the five problems from the VSTTE 2010 verification competition [11]. The source code for all these examples is contained in Why3's distribution, in sub-directory `examples/`. Look for files `logic/einstein.why` and `vstte10_xxx.mlw`.

2.1 Problem 0: Einstein's Problem

Let us use Why3 to solve a little puzzle known as "Einstein's logic problem".[1] The problem is stated as follows. Five persons, of five different nationalities, live in five houses in a row, all painted with different colors. These five persons own different pets, drink different beverages, and smoke different brands of cigars. We are given the following information:

- The Englishman lives in a red house;
- The Swede has dogs;
- The Dane drinks tea;
- The green house is on the left of the white one;
- The green house's owner drinks coffee;
- The person who smokes Pall Mall has birds;
- The yellow house's owner smokes Dunhill;
- In the house in the center lives someone who drinks milk;
- The Norwegian lives in the first house;
- The man who smokes Blends lives next to the one who has cats;
- The man who owns a horse lives next to the one who smokes Dunhills;
- The man who smokes Blue Masters drinks beer;
- The German smokes Prince;
- The Norwegian lives next to the blue house;
- The man who smokes Blends has a neighbour who drinks water.

The question is: what is the nationality of the fish's owner?

We start by introducing a general-purpose theory defining the notion of *bijection*, as two abstract types together with two functions from one to the other and two axioms stating that these functions are inverse of each other.

[1] This Why3 example was contributed by Stéphane Lescuyer.

```
theory Bijection
  type t
  type u

  function of t : u
  function to_ u : t

  axiom To_of : forall x : t. to_ (of x) = x
  axiom Of_to : forall y : u. of (to_ y) = y
end
```

We now start a new theory, `Einstein`, which will contain all the individuals of the problem.

```
theory Einstein
```

First, we introduce enumeration types for houses, colors, persons, drinks, cigars, and pets.

```
  type house  = H1 | H2 | H3 | H4 | H5
  type color  = Blue | Green | Red | White | Yellow
  type person = Dane | Englishman | German | Norwegian | Swede
  type drink  = Beer | Coffee | Milk | Tea | Water
  type cigar  = Blend | BlueMaster | Dunhill | PallMall | Prince
  type pet    = Birds | Cats | Dogs | Fish | Horse
```

We now express that each house is associated bijectively to a color, by *cloning* the `Bijection` theory appropriately.

```
  clone Bijection as Color with type t = house, type u = color
```

Cloning a theory makes a copy of all its declarations, possibly in combination with a user-provided substitution. Here we substitute type `house` for type `t` and type `color` for type `u`. As a result, we get two new functions, namely `Color.of` and `Color.to_`, from houses to colors and colors to houses, respectively, and two new axioms relating them. Similarly, we express that each house is associated bijectively to a person

```
  clone Bijection as Owner with type t = house, type u = person
```

and that drinks, cigars, and pets are all associated bijectively to persons:

```
  clone Bijection as Drink with type t = person, type u = drink
  clone Bijection as Cigar with type t = person, type u = cigar
  clone Bijection as Pet   with type t = person, type u = pet
```

Next, we need a way to state that a person lives next to another. We first define a predicate `leftof` over two houses.

```
  predicate leftof (h1 h2 : house) =
    match h1, h2 with
    | H1, H2
    | H2, H3
    | H3, H4
    | H4, H5 -> true
    | _      -> false
    end
```

Note how we advantageously used pattern matching, with an or-pattern for the four positive cases and a universal pattern for the remaining 21 cases. It is then immediate to define a `neighbour` predicate over two houses, which completes theory `Einstein`.

```
  predicate rightof (h1 h2 : house) =
    leftof h2 h1
  predicate neighbour (h1 h2 : house) =
    leftof h1 h2 \/ rightof h1 h2
end
```

The next theory contains the 15 hypotheses. It starts by importing theory `Einstein`.

```
theory EinsteinHints
  use import Einstein
```

Then each hypothesis is stated in terms of `to_` and `of` functions. For instance, the hypothesis "The Englishman lives in a red house" is declared as the following axiom.

```
  axiom Hint1: Color.of (Owner.to_ Englishman) = Red
```

And so on for all other hypotheses, up to "The man who smokes Blends has a neighbour who drinks water", which completes this theory.

```
  ...
  axiom Hint15:
    neighbour (Owner.to_ (Cigar.to_ Blend)) (Owner.to_ (Drink.to_ Water))
end
```

Finally, we declare the goal in a fourth theory:

```
theory Problem
  use import Einstein
  use import EinsteinHints

  goal G: Pet.to_ Fish = German
end
```

and we can use Why3 to discharge this goal with any prover of our choice.

```
> why3 prove -P alt-ergo einstein.why
einstein.why Goals G: Valid (1.27s, 989 steps)
```

The source code for this puzzle is available in the source distribution of Why3, in file `examples/logic/einstein.why`.

2.2 Problem 1: Sum and Maximum

Let us now move to the problems of the VSTTE 2010 verification competition [11]. The first problem is stated as follows:

> Given an N-element array of natural numbers, write a program to compute the sum and the maximum of the elements in the array.

We assume $N \geq 0$ and $a[i] \geq 0$ for $0 \leq i < N$, as precondition, and we have to prove the following postcondition:

$$sum \leq N \times max.$$

In a file `max_sum.mlw`, we start a new module:

```
module MaxAndSum
```

We are obviously needing arithmetic, so we import the corresponding theory, exactly as we would do within a theory definition:

```
  use import int.Int
```

We are also going to use references and arrays from Why3 standard library, so we import the corresponding modules:

```
  use import ref.Ref
  use import array.Array
```

Modules `Ref` and `Array` respectively provide a type `ref 'a` for references and a type `array 'a` for arrays, together with useful operations and traditional syntax. They are loaded from the WhyML files `ref.mlw` and `array.mlw` in the standard library.

We are now in position to define a program function `max_sum`. A function definition is introduced with the keyword `let`. In our case, it introduces a function with two arguments, an array `a` and its size `n`:

```
  let max_sum (a: array int) (n: int) : (int, int) = ...
```

(There is a function `length` to get the size of an array but we add this extra parameter `n` to stay close to the original problem statement.) The function body is a Hoare triple, that is a precondition, a program expression, and a postcondition.

```
  let max_sum (a: array int) (n: int) : (int, int)
    requires { n = length a }
    requires { forall i. 0 <= i < n -> a[i] >= 0 }
    ensures  { let (sum, max) = result in sum <= n * max }
  = ... expression ...
```

The first precondition expresses that `n` is equal to the length of `a` (this will be needed for verification conditions related to array bound checking). The second precondition expresses that all elements of `a` are non-negative. The postcondition decomposes the value returned by the function as a pair of integers `(sum, max)` and states the required property.

```
    returns { sum, max -> sum <= n * max }
```

We are now left with the function body itself, that is a code computing the sum and the maximum of all elements in `a`. With no surprise, it is as simple as introducing two local references

```
    let sum = ref 0 in
    let max = ref 0 in
```

scanning the array with a `for` loop, updating `max` and `sum`

```
    for i = 0 to n - 1 do
      if !max < a[i] then max := a[i];
      sum := !sum + a[i]
    done;
```

and finally returning the pair of the values contained in `sum` and `max`:

```
module MaxAndSum

  use import int.Int
  use import ref.Ref
  use import array.Array

  let max_sum (a: array int) (n: int) : (int, int)
    requires { n = length a }
    requires { forall i. 0 <= i < n -> a[i] >= 0 }
    returns  { sum, max -> sum <= n * max }
  = let sum = ref 0 in
    let max = ref 0 in
    for i = 0 to n - 1 do
      invariant { !sum <= i * !max }
      if !max < a[i] then max := a[i];
      sum := !sum + a[i]
    done;
    !sum, !max

end
```

Figure 2.1: Solution for VSTTE'10 competition problem 1.

```
  !sum, !max
```

This completes the code for function `max_sum`. As such, it cannot be proved correct, since the loop is still lacking a loop invariant. In this case, the loop invariant is as simple as `!sum <= i * !max`, since the postcondition only requires us to prove `sum <= n * max`. The loop invariant is introduced with the keyword `invariant`, immediately after the keyword `do`:

```
    for i = 0 to n - 1 do
      invariant { !sum <= i * !max }
      ...
    done
```

There is no need to introduce a variant, as the termination of a `for` loop is automatically guaranteed. This completes module `MaxAndSum`. Figure 2.1 shows the whole code. We can now proceed to its verification. Running `why3`, or better `why3 ide`, on file `max_sum.mlw` shows a single verification condition with name `WP max_sum`. Discharging this verification condition requires a little bit of non-linear arithmetic. Thus some SMT solvers may fail at proving it, but other succeed, *e.g.*, CVC4.

2.3 Problem 2: Inverting an Injection

The second problem is stated as follows:

> Invert an injective array A on N elements in the subrange from 0 to $N-1$, *i.e.* the output array B must be such that $B[A[i]] = i$ for $0 \leq i < N$.

The code is immediate, since it is as simple as

```
    for i = 0 to n - 1 do b[a[i]] <- i done
```

so it is more a matter of specification and of getting the proof done with as much automation as possible. In a new file, we start a new module and we import arithmetic and arrays:

```
module InvertingAnInjection
  use import int.Int
  use import array.Array
```

It is convenient to introduce predicate definitions for the properties of being injective and surjective. These are purely logical declarations:

```
  predicate injective (a: array int) (n: int) =
    forall i j. 0 <= i < n -> 0 <= j < n -> i <> j -> a[i] <> a[j]

  predicate surjective (a: array int) (n: int) =
    forall i. 0 <= i < n -> exists j: int. (0 <= j < n /\ a[j] = i)
```

It is also convenient to introduce the predicate "being in the subrange from 0 to $n-1$":

```
  predicate range (a: array int) (n: int) =
    forall i. 0 <= i < n -> 0 <= a[i] < n
```

Using these predicates, we can formulate the assumption that any injective array of size n within the range $0..n-1$ is also surjective:

```
  lemma injective_surjective:
    forall a: array int, n: int.
      injective a n -> range a n -> surjective a n
```

We declare it as a lemma rather than as an axiom, since it is actually provable. It requires induction and can be proved using the Coq proof assistant for instance. Finally we can give the code a specification, with a loop invariant which simply expresses the values assigned to array `b` so far:

```
  let inverting (a: array int) (b: array int) (n: int)
    requires { n = length a = length b }
    requires { injective a n /\ range a n }
    ensures  { injective b n }
  = for i = 0 to n - 1 do
      invariant { forall j. 0 <= j < i -> b[a[j]] = j }
      b[a[i]] <- i
    done
```

Here we chose to have array `b` as argument; returning a freshly allocated array would be equally simple. The whole module is given in Figure 2.2. The verification conditions for function `inverting` are easily discharged automatically, thanks to the lemma.

2.4 Problem 3: Searching a Linked List

The third problem is stated as follows:

> Given a linked list representation of a list of integers, find the index of the first element that is equal to 0.

```
module InvertingAnInjection

  use import int.Int
  use import array.Array

  predicate injective (a: array int) (n: int) =
    forall i j. 0 <= i < n -> 0 <= j < n -> i <> j -> a[i] <> a[j]

  predicate surjective (a: array int) (n: int) =
    forall i. 0 <= i < n -> exists j: int. (0 <= j < n /\ a[j] = i)

  predicate range (a: array int) (n: int) =
    forall i. 0 <= i < n -> 0 <= a[i] < n

  lemma injective_surjective:
    forall a: array int, n: int.
      injective a n -> range a n -> surjective a n

  let inverting (a: array int) (b: array int) (n: int)
    requires { n = length a = length b }
    requires { injective a n /\ range a n }
    ensures  { injective b n }
  = for i = 0 to n - 1 do
      invariant { forall j. 0 <= j < i -> b[a[j]] = j }
      b[a[i]] <- i
    done

end
```

Figure 2.2: Solution for VSTTE'10 competition problem 2.

More precisely, the specification says

> You have to show that the program returns an index i equal to the length of the list if there is no such element. Otherwise, the i-th element of the list must be equal to 0, and all the preceding elements must be non-zero.

Since the list is not mutated, we can use the algebraic data type of polymorphic lists from Why3's standard library, defined in theory `list.List`. It comes with other handy theories: `list.Length`, which provides a function `length`, and `list.Nth`, which provides a function `nth` for the n-th element of a list. The latter returns an option type, depending on whether the index is meaningful or not.

```
module SearchingALinkedList
  use import int.Int
  use import option.Option
  use export list.List
  use export list.Length
  use export list.Nth
```

It is helpful to introduce two predicates: a first one for a successful search,

```
predicate zero_at (l: list int) (i: int) =
  nth i l = Some 0 /\ forall j. 0 <= j < i -> nth j l <> Some 0
```

and a second one for a non-successful search,

```
predicate no_zero (l: list int) =
  forall j. 0 <= j < length l -> nth j l <> Some 0
```

We are now in position to give the code for the search function. We write it as a recursive function `search` that scans a list for the first zero value:

```
let rec search (i: int) (l: list int) : int =
  match l with
  | Nil      -> i
  | Cons x r -> if x = 0 then i else search (i+1) r
  end
```

Passing an index `i` as first argument allows to perform a tail call. A simpler code (yet less efficient) would return 0 in the first branch and `1 + search ...` in the second one, avoiding the extra argument `i`.

We first prove the termination of this recursive function. It amounts to give it a *variant*, that is a value that strictly decreases at each recursive call with respect to some well-founded ordering. Here it is as simple as the list `l` itself:

```
let rec search (i: int) (l: list int) : int variant { l } = ...
```

It is worth pointing out that variants are not limited to values of algebraic types. A non-negative integer term (for example, `length l`) can be used, or a term of any other type equipped with a well-founded order relation. Several terms can be given, separated with commas, for lexicographic ordering.

There is no precondition for function `search`. The postcondition expresses that either a zero value is found, and consequently the value returned is bounded accordingly,

```
i <= result < i + length l /\ zero_at l (result - i)
```

or no zero value was found, and thus the returned value is exactly `i` plus the length of `l`:

```
result = i + length l /\ no_zero l
```

Solving the problem is simply a matter of calling `search` with 0 as first argument. The code is given Figure 2.3. The verification conditions are all discharged automatically.

Alternatively, we can implement the search with a `while` loop. To do this, we need to import references from the standard library, together with theory `list.HdTl` which defines functions `hd` and `tl` over lists.

```
use import ref.Ref
use import list.HdTl
```

Being partial functions, `hd` and `tl` return options. For the purpose of our code, though, it is simpler to have functions which do not return options, but have preconditions instead. Such a function `head` is defined as follows:

```
let head (l: list 'a) : 'a
  requires { l <> Nil } ensures { hd l = Some result }
= match l with Nil -> absurd | Cons h _ -> h end
```

```
module SearchingALinkedList

  use import int.Int
  use export list.List
  use export list.Length
  use export list.Nth

  predicate zero_at (l: list int) (i: int) =
    nth i l = Some 0 /\ forall j. 0 <= j < i -> nth j l <> Some 0

  predicate no_zero (l: list int) =
    forall j. 0 <= j < length l -> nth j l <> Some 0

  let rec search (i: int) (l: list int) : int variant { l }
    ensures { (i <= result < i + length l /\ zero_at l (result - i))
           \/ (result = i + length l /\ no_zero l) }
  = match l with
    | Nil -> i
    | Cons x r -> if x = 0 then i else search (i+1) r
    end

  let search_list (l: list int) : int
    ensures { (0 <= result < length l /\ zero_at l result)
           \/ (result = length l /\ no_zero l) }
  = search 0 l

end
```

Figure 2.3: Solution for VSTTE'10 competition problem 3.

The program construct `absurd` denotes an unreachable piece of code. It generates the verification condition `false`, which is here provable using the precondition (the list cannot be `Nil`). Function `tail` is defined similarly:

```
  let tail (l: list 'a) : list 'a
    requires { l <> Nil } ensures { tl l = Some result }
  = match l with Nil -> absurd | Cons _ t -> t end
```

Using `head` and `tail`, it is straightforward to implement the search as a `while` loop. It uses a local reference `i` to store the index and another local reference `s` to store the list being scanned. As long as `s` is not empty and its head is not zero, it increments `i` and advances in `s` using function `tail`.

```
  let search_loop (l: list int) : int =
    ensures { ... same postcondition as in search_list ... }
  = let i = ref 0 in
    let s = ref l in
    while !s <> Nil && head !s <> 0 do
      invariant { ... }
      variant   { !s }
      i := !i + 1;
```

```
    s := tail !s
  done;
  !i
```

The postcondition is exactly the same as for function `search_list`. The termination of the `while` loop is ensured using a variant, exactly as for a recursive function. Such a variant must strictly decrease at each execution of the loop body. The reader is invited to figure out the loop invariant.

2.5 Problem 4: N-Queens

The fourth problem is probably the most challenging one. We have to verify the implementation of a program which solves the N-queens puzzle: place N queens on an $N \times N$ chess board so that no queen can capture another one with a legal move. The program should return a placement if there is a solution and indicates that there is no solution otherwise. A placement is a N-element array which assigns the queen on row i to its column. Thus we start our module by importing arithmetic and arrays:

```
module NQueens
  use import int.Int
  use import array.Array
```

The code is a simple backtracking algorithm, which tries to put a queen on each row of the chess board, one by one (there is basically no better way to solve the N-queens puzzle). A building block is a function which checks whether the queen on a given row may attack another queen on a previous row. To verify this function, we first define a more elementary predicate, which expresses that queens on row `pos` and `q` do no attack each other:

```
  predicate consistent_row (board: array int) (pos: int) (q: int) =
    board[q] <> board[pos] /\
    board[q] - board[pos] <> pos - q /\
    board[pos] - board[q] <> pos - q
```

Then it is possible to define the consistency of row `pos` with respect to all previous rows:

```
  predicate is_consistent (board: array int) (pos: int) =
    forall q. 0 <= q < pos -> consistent_row board pos q
```

Implementing a function which decides this predicate is another matter. In order for it to be efficient, we want to return `False` as soon as a queen attacks the queen on row `pos`. We use an exception for this purpose and it carries the row of the attacking queen:

```
  exception Inconsistent int
```

The check is implemented by a function `check_is_consistent`, which takes the board and the row `pos` as arguments, and scans rows from 0 to `pos-1` looking for an attacking queen. As soon as one is found, the exception is raised. It is caught immediately outside the loop and `False` is returned. Whenever the end of the loop is reached, `True` is returned.

```
  let check_is_consistent (board: array int) (pos: int) : bool
    requires { 0 <= pos < length board }
    ensures  { result <-> is_consistent board pos }
  = try
      for q = 0 to pos - 1 do
```

```
      invariant {
        forall j:int. 0 <= j < q -> consistent_row board pos j
      }
      let bq   = board[q]   in
      let bpos = board[pos] in
      if bq        = bpos    then raise (Inconsistent q);
      if bq - bpos = pos - q then raise (Inconsistent q);
      if bpos - bq = pos - q then raise (Inconsistent q)
    done;
    True
  with Inconsistent q ->
    assert { not (consistent_row board pos q) };
    False
  end
```

The assertion in the exception handler is a cut for SMT solvers. This first part of the solution is given in Figure 2.4.

We now proceed with the verification of the backtracking algorithm. The specification requires us to define the notion of solution, which is straightforward using the predicate `is_consistent` above. However, since the algorithm will try to complete a given partial solution, it is more convenient to define the notion of partial solution, up to a given row. It is even more convenient to split it in two predicates, one related to legal column values and another to consistency of rows:

```
predicate is_board (board: array int) (pos: int) =
  forall q. 0 <= q < pos -> 0 <= board[q] < length board

predicate solution (board: array int) (pos: int) =
  is_board board pos /\
  forall q. 0 <= q < pos -> is_consistent board q
```

The algorithm will not mutate the partial solution it is given and, in case of a search failure, will claim that there is no solution extending this prefix. For this reason, we introduce a predicate comparing two chess boards for equality up to a given row:

```
predicate eq_board (b1 b2: array int) (pos: int) =
  forall q. 0 <= q < pos -> b1[q] = b2[q]
```

The search itself makes use of an exception to signal a successful search:

```
exception Solution
```

The backtracking code is a recursive function `bt_queens` which takes the chess board, its size, and the starting row for the search. The termination is ensured by the obvious variant `n-pos`.

```
let rec bt_queens (board: array int) (n: int) (pos: int) : unit
  variant  { n - pos }
```

The precondition relates `board`, `pos`, and `n` and requires `board` to be a solution up to `pos`:

```
  requires { 0 <= pos <= n = length board }
  requires { solution board pos }
```

```
module NQueens
  use import int.Int
  use import array.Array

  predicate consistent_row (board: array int) (pos: int) (q: int) =
    board[q] <> board[pos] /\
    board[q] - board[pos] <> pos - q /\
    board[pos] - board[q] <> pos - q

  predicate is_consistent (board: array int) (pos: int) =
    forall q. 0 <= q < pos -> consistent_row board pos q

  exception Inconsistent int

  let check_is_consistent (board: array int) (pos: int)
    requires { 0 <= pos < length board }
    ensures  { result <-> is_consistent board pos }
  = try
      for q = 0 to pos - 1 do
        invariant {
          forall j:int. 0 <= j < q -> consistent_row board pos j
        }
        let bq   = board[q]   in
        let bpos = board[pos] in
        if bq        = bpos    then raise (Inconsistent q);
        if bq - bpos = pos - q then raise (Inconsistent q);
        if bpos - bq = pos - q then raise (Inconsistent q)
      done;
      True
    with Inconsistent q ->
      assert { not (consistent_row board pos q) };
      False
    end
```

Figure 2.4: Solution for VSTTE'10 competition problem 4 (1/2).

The postcondition is twofold: either the function exits normally and then there is no solution extending the prefix in `board`, which has not been modified; or the function raises `Solution` and we have a solution in `board`.

```
    ensures  { eq_board board (old board) pos }
    ensures  { forall b:array int. length b = n -> is_board b n ->
                 eq_board board b pos -> not (solution b n) }
    raises   { Solution -> solution board n }
  =
```

Whenever we reach the end of the chess board, we have found a solution and we signal it using exception `Solution`:

```
    if pos = n then raise Solution;
```

Otherwise we scan all possible positions for the queen on row `pos` with a `for` loop:

```
    for i = 0 to n - 1 do
```

The loop invariant states that we have not modified the solution prefix so far, and that we have not found any solution that would extend this prefix with a queen on row `pos` at a column below `i`:

```
      invariant { eq_board board (old board) pos }
      invariant { forall b:array int.  length b = n -> is_board b n ->
        eq_board board b pos -> 0 <= b[pos] < i -> not (solution b n) }
```

Then we assign column `i` to the queen on row `pos` and we check for a possible attack with `check_is_consistent`. If not, we call `bt_queens` recursively on the next row.

```
      board[pos] <- i;
      if check_is_consistent board pos then bt_queens board n (pos + 1)
    done
```

This completes the loop and function `bt_queens` as well. Solving the puzzle is a simple call to `bt_queens`, starting the search on row 0. The postcondition is also twofold, as for `bt_queens`, yet slightly simpler.

```
  let queens (board: array int) (n: int) : unit
    requires { length board = n }
    ensures  { forall b:array int.
                 length b = n -> is_board b n -> not (solution b n) }
    raises   { Solution -> solution board n }
  = bt_queens board n 0
```

This second part of the solution is given Figure 2.5. With the help of a few auxiliary lemmas — not given here but available from Why3's sources — the verification conditions are all discharged automatically, including the verification of the lemmas themselves.

2.6 Problem 5: Amortized Queue

The last problem consists in verifying the implementation of a well-known purely applicative data structure for queues. A queue is composed of two lists, *front* and *rear*. We push elements at the head of list *rear* and pop them off the head of list *front*. We maintain that the length of *front* is always greater or equal to the length of *rear*. (See for instance Okasaki's *Purely Functional Data Structures* [9] for more details.)

We have to implement operations `empty`, `head`, `tail`, and `enqueue` over this data type, to show that the invariant over lengths is maintained, and finally

> to show that a client invoking these operations observes an abstract queue given by a sequence.

In a new module, we import arithmetic and theory `list.ListRich`, a combo theory that imports all list operations we will require: length, reversal, and concatenation.

```
module AmortizedQueue
  use import int.Int
  use import option.Option
  use export list.ListRich
```

```
  predicate is_board (board: array int) (pos: int) =
    forall q. 0 <= q < pos -> 0 <= board[q] < length board

  predicate solution (board: array int) (pos: int) =
    is_board board pos /\
    forall q. 0 <= q < pos -> is_consistent board q

  predicate eq_board (b1 b2: array int) (pos: int) =
    forall q. 0 <= q < pos -> b1[q] = b2[q]

  exception Solution

  let rec bt_queens (board: array int) (n: int) (pos: int) : unit
    variant  { n - pos }
    requires { 0 <= pos <= n = length board }
    requires { solution board pos }
    ensures  { eq_board board (old board) pos }
    ensures  { forall b:array int. length b = n -> is_board b n ->
                 eq_board board b pos -> not (solution b n) }
    raises   { Solution -> solution board n }
  = if pos = n then raise Solution;
    for i = 0 to n - 1 do
      invariant { eq_board board (old board) pos }
      invariant { forall b:array int. length b = n -> is_board b n ->
        eq_board board b pos -> 0 <= b[pos] < i -> not (solution b n) }
      board[pos] <- i;
      if check_is_consistent board pos then bt_queens board n (pos + 1)
    done

  let queens (board: array int) (n: int) : unit
    requires { length board = n }
    ensures  { forall b:array int.
                 length b = n -> is_board b n -> not (solution b n) }
    raises   { Solution -> solution board n }
  = bt_queens board n 0

end
```

Figure 2.5: Solution for VSTTE'10 competition problem 4 (2/2).

The queue data type is naturally introduced as a polymorphic record type. The two list lengths are explicitly stored, for greater efficiency.

```
type queue 'a = { front: list 'a; lenf: int;
                  rear : list 'a; lenr: int; }
invariant { length front = lenf >= length rear = lenr }
by { front = Nil; lenf = 0; rear = Nil; lenr = 0 }
```

The type definition is accompanied with an invariant — a logical property imposed on any value of the type. Why3 assumes that any `queue` passed as an argument to a program function satisfies the invariant and it produces a proof obligation every time a `queue` is created. The `by` clause ensures the non-vacuity of this type with invariant. If you omit it, a goal with an existential statement is generated.

For the purpose of the specification, it is convenient to introduce a function `sequence` which builds the sequence of elements of a queue, that is the front list concatenated to the reversed rear list.

```
function sequence (q: queue 'a) : list 'a = q.front ++ reverse q.rear
```

It is worth pointing out that this function can only be used in specifications. We start with the easiest operation: building the empty queue.

```
let empty () : queue 'a
  ensures { sequence result = Nil }
= { front = Nil; lenf = 0; rear = Nil; lenr = 0 }
```

The postcondition states that the returned queue represents the empty sequence. Another postcondition, saying that the returned queue satisfies the type invariant, is implicit. Note the cast to type `queue 'a`. It is required, for the type checker not to complain about an undefined type variable.

The next operation is `head`, which returns the first element from a given queue `q`. It naturally requires the queue to be non empty, which is conveniently expressed as `sequence q` not being `Nil`.

```
let head (q: queue 'a) : 'a
  requires { sequence q <> Nil }
  ensures  { hd (sequence q) = Some result }
= let Cons x _ = q.front in x
```

The fact that the argument `q` satisfies the type invariant is implicitly assumed. The type invariant is required to prove the absurdity of `q.front` being `Nil` (otherwise, `sequence q` would be `Nil` as well).

The next operation is `tail`, which removes the first element from a given queue. This is more subtle than `head`, since we may have to re-structure the queue to maintain the invariant. Since we will have to perform a similar operation when implementing operation `enqueue` later, it is a good idea to introduce a smart constructor `create` that builds a queue from two lists while ensuring the invariant. The list lengths are also passed as arguments, to avoid unnecessary computations.

```
let create (f: list 'a) (lf: int) (r: list 'a) (lr: int) : queue 'a
  requires { lf = length f /\ lr = length r }
  ensures  { sequence result = f ++ reverse r }
= if lf >= lr then
    { front = f; lenf = lf; rear = r; lenr = lr }
```

```
    else
      let f = f ++ reverse r in
      { front = f; lenf = lf + lr; rear = Nil; lenr = 0 }
```

If the invariant already holds, it is simply a matter of building the record. Otherwise, we empty the rear list and build a new front list as the concatenation of list `f` and the reversal of list `r`. The principle of this implementation is that the cost of this reversal will be amortized over all queue operations. Implementing function `tail` is now straightforward and follows the structure of function `head`.

```
let tail (q: queue 'a) : queue 'a
  requires { sequence q <> Nil }
  ensures  { tl (sequence q) = Some (sequence result) }
= let Cons _ r = q.front in
  create r (q.lenf - 1) q.rear q.lenr
```

The last operation is `enqueue`, which pushes a new element in a given queue. Reusing the smart constructor `create` makes it a one line code.

```
let enqueue (x: 'a) (q: queue 'a) : queue 'a
  ensures { sequence result = sequence q ++ Cons x Nil }
= create q.front q.lenf (Cons x q.rear) (q.lenr + 1)
```

The code is given Figure 2.6. The verification conditions are all discharged automatically.

```
module AmortizedQueue

  use import int.Int
  use import option.Option
  use import list.ListRich

  type queue 'a = { front: list 'a; lenf: int;
                    rear : list 'a; lenr: int; }
    invariant { length front = lenf >= length rear = lenr }
    by { front = Nil; lenf = 0; rear = Nil; lenr = 0 }

  function sequence (q: queue 'a) : list 'a =
    q.front ++ reverse q.rear

  let empty () : queue 'a
    ensures { sequence result = Nil }
  = { front = Nil; lenf = 0; rear = Nil; lenr = 0 }

  let head (q: queue 'a) : 'a
    requires { sequence q <> Nil }
    ensures  { hd (sequence q) = Some result }
  = let Cons x _ = q.front in x

  let create (f: list 'a) (lf: int) (r: list 'a) (lr: int) : queue 'a
    requires { lf = length f /\ lr = length r }
    ensures  { sequence result = f ++ reverse r }
  = if lf >= lr then
      { front = f; lenf = lf; rear = r; lenr = lr }
    else
      let f = f ++ reverse r in
      { front = f; lenf = lf + lr; rear = Nil; lenr = 0 }

  let tail (q: queue 'a) : queue 'a
    requires { sequence q <> Nil }
    ensures  { tl (sequence q) = Some (sequence result) }
  = let Cons _ r = q.front in
    create r (q.lenf - 1) q.rear q.lenr

  let enqueue (x: 'a) (q: queue 'a) : queue 'a
    ensures { sequence result = sequence q ++ Cons x Nil }
  = create q.front q.lenf (Cons x q.rear) (q.lenr + 1)

end
```

Figure 2.6: Solution for VSTTE'10 competition problem 5.

Chapter 3

The Why3 API

This chapter is a tutorial for the users who want to link their own OCaml code with the Why3 library. We progressively introduce the way one can use the library to build terms, formulas, theories, proof tasks, call external provers on tasks, and apply transformations on tasks. The complete documentation for API calls is given at URL http://why3.lri.fr/api-1.2.1/.

We assume the reader has a fair knowledge of the OCaml language. Notice that the Why3 library must be installed, see Section 4.1.2. The OCaml code given below is available in the source distribution in directory `examples/use_api/` together with a few other examples.

3.1 Building Propositional Formulas

The first step is to know how to build propositional formulas. The module `Term` gives a few functions for building these. Here is a piece of OCaml code for building the formula $true \vee false$.

```
(* opening the Why3 library *)
open Why3

(* a ground propositional goal: true or false *)
let fmla_true : Term.term = Term.t_true
let fmla_false : Term.term = Term.t_false
let fmla1 : Term.term = Term.t_or fmla_true fmla_false
```

The library uses the common type `term` both for terms (*i.e.* expressions that produce a value of some particular type) and formulas (*i.e.* boolean-valued expressions).

Such a formula can be printed using the module `Pretty` providing pretty-printers.

```
(* printing it *)
open Format
let () = printf "@[formula 1 is:@ %a@]@." Pretty.print_term fmla1
```

Assuming the lines above are written in a file `f.ml`, it can be compiled using

```
ocamlfind ocamlc -package why3 -linkpkg f.ml -o f
```

Running the generated executable `f` results in the following output.

```
formula 1 is: true \/ false
```

Let us now build a formula with propositional variables: $A \wedge B \rightarrow A$. Propositional variables must be declared first before using them in formulas. This is done as follows.

```
let prop_var_A : Term.lsymbol =
  Term.create_psymbol (Ident.id_fresh "A") []
let prop_var_B : Term.lsymbol =
  Term.create_psymbol (Ident.id_fresh "B") []
```

The type `lsymbol` is the type of function and predicate symbols (which we call logic symbols for brevity). Then the atoms A and B must be built by the general function for applying a predicate symbol to a list of terms. Here we just need the empty list of arguments.

```
let atom_A : Term.term = Term.ps_app prop_var_A []
let atom_B : Term.term = Term.ps_app prop_var_B []
let fmla2 : Term.term =
  Term.t_implies (Term.t_and atom_A atom_B) atom_A
let () = printf "@[formula 2 is:@ %a@]@." Pretty.print_term fmla2
```

As expected, the output is as follows.

```
formula 2 is: A /\ B -> A
```

Notice that the concrete syntax of Why3 forbids function and predicate names to start with a capital letter (except for the algebraic type constructors which must start with one). This constraint is not enforced when building those directly using library calls.

3.2 Building Tasks

Let us see how we can call a prover to prove a formula. As said in previous chapters, a prover must be given a task, so we need to build tasks from our formulas. Task can be build incrementally from an empty task by adding declaration to it, using the functions `add_*_decl` of module `Task`. For the formula $true \vee false$ above, this is done as follows.

```
(* building the task for formula 1 alone *)
let task1 : Task.task = None (* empty task *)
let goal_id1 : Decl.prsymbol = Decl.create_prsymbol (Ident.id_fresh "goal1")
let task1 : Task.task = Task.add_prop_decl task1 Decl.Pgoal goal_id1 fmla1
```

To make the formula a goal, we must give a name to it, here "goal1". A goal name has type `prsymbol`, for identifiers denoting propositions in a theory or a task. Notice again that the concrete syntax of Why3 requires these symbols to be capitalized, but it is not mandatory when using the library. The second argument of `add_prop_decl` is the kind of the proposition: `Paxiom`, `Plemma` or `Pgoal`. Notice that lemmas are not allowed in tasks and can only be used in theories.

Once a task is built, it can be printed.

```
(* printing the task *)
let () = printf "@[task 1 is:@\n%a@]@." Pretty.print_task task1
```

The task for our second formula is a bit more complex to build, because the variables A and B must be added as abstract (*i.e.* not defined) propositional symbols in the task.

```
(* task for formula 2 *)
let task2 = None
let task2 = Task.add_param_decl task2 prop_var_A
let task2 = Task.add_param_decl task2 prop_var_B
let goal_id2 = Decl.create_prsymbol (Ident.id_fresh "goal2")
let task2 = Task.add_prop_decl task2 Decl.Pgoal goal_id2 fmla2
let () = printf "@[task 2 created:@\n%a@]@." Pretty.print_task task2
```

Execution of our OCaml program now outputs:

```
task 1 is:
theory Task
  goal Goal1 : true \/ false
end

task 2 is:
theory Task
  predicate A

  predicate B

  goal Goal2 : A /\ B -> A
end
```

3.3 Calling External Provers

To call an external prover, we need to access the Why3 configuration file `why3.conf`, as it was built using the `why3config` command line tool or the Detect Provers menu of the graphical IDE. The following API calls allow to access the content of this configuration file.

```
(* reads the config file *)
let config : Whyconf.config = Whyconf.read_config None
(* the [main] section of the config file *)
let main : Whyconf.main = Whyconf.get_main config
(* all the provers detected, from the config file *)
let provers : Whyconf.config_prover Whyconf.Mprover.t =
  Whyconf.get_provers config
```

The type `'a Whyconf.Mprover.t` is a map indexed by provers. A prover is a record with a name, a version, and an alternative description (to differentiate between various configurations of a given prover). Its definition is in the module `Whyconf`:

```
type prover =
   { prover_name : string;
     prover_version : string;
     prover_altern : string;
   }
```

The map `provers` provides the set of existing provers. In the following, we directly attempt to access a prover named “Alt-Ergo”, any version.

```
(* One prover named Alt-Ergo in the config file *)
let alt_ergo : Whyconf.config_prover =
  let fp = Whyconf.parse_filter_prover "Alt-Ergo" in
  (** all provers that have the name "Alt-Ergo" *)
  let provers = Whyconf.filter_provers config fp in
  if Whyconf.Mprover.is_empty provers then begin
    eprintf "Prover Alt-Ergo not installed or not configured@.";
    exit 0
  end else
    snd (Whyconf.Mprover.max_binding provers)
```

We could also get a specific version with :

```
(* Specific version 2.0.0 of Alt-Ergo in the config file *)
let alt_ergo_2_0_0 : Whyconf.config_prover =
  let fp = Whyconf.parse_filter_prover "Alt-Ergo,2.0.0" in
  let provers = Whyconf.filter_provers config fp in
  if Whyconf.Mprover.is_empty provers then begin
    eprintf "Prover Alt-Ergo 2.0.0 not installed or not configured@.";
    exit 0
  end else
    snd (Whyconf.Mprover.max_binding provers)
```

The next step is to obtain the driver associated to this prover. A driver typically depends on the standard theories so these should be loaded first.

```
(* builds the environment from the [loadpath] *)
let env : Env.env = Env.create_env (Whyconf.loadpath main)

(* loading the Alt-Ergo driver *)
let alt_ergo_driver : Driver.driver =
  try
    Whyconf.load_driver main env alt_ergo.Whyconf.driver []
  with e ->
    eprintf "Failed to load driver for alt-ergo: %a@."
      Exn_printer.exn_printer e;
    exit 1
```

We are now ready to call the prover on the tasks. This is done by a function call that launches the external executable and waits for its termination. Here is a simple way to proceed:

```
(* calls Alt-Ergo *)
let result1 : Call_provers.prover_result =
  Call_provers.wait_on_call
    (Driver.prove_task ~limit:Call_provers.empty_limit
                       ~command:alt_ergo.Whyconf.command
    alt_ergo_driver task1)

(* prints Alt-Ergo answer *)
let () = printf "@[On task 1, Alt-Ergo answers %a@."
  Call_provers.print_prover_result result1
```

This way to call a prover is in general too naive, since it may never return if the prover runs without time limit. The function `prove_task` has an optional parameter `limit`, a record defined in module `Call_provers`:

```
type resource_limit = {
  limit_time : int;
  limit_mem : int;
  limit_steps : int;
}
```

where the field `limit_time` is the maximum allowed running time in seconds, and `limit_mem` is the maximum allowed memory in megabytes. The type `prover_result` is a record defined in module `Call_provers`:

```
type prover_result = {
  pr_answer : prover_answer;
  pr_status : Unix.process_status;
  pr_output : string;
  pr_time : float;
  pr_steps : int;  (* -1 if unknown *)
  pr_model : model;
}
```

with in particular the fields:

- `pr_answer`: the prover answer, explained below;
- `pr_time` : the time taken by the prover, in seconds.

A `pr_answer` is the sum type defined in module `Call_provers`:

```
type prover_answer =
  | Valid
  | Invalid
  | Timeout
  | OutOfMemory
  | StepLimitExceeded
  | Unknown of string
  | Failure of string
  | HighFailure
```

corresponding to these kinds of answers:

- `Valid`: the task is valid according to the prover.
- `Invalid`: the task is invalid.
- `Timeout`: the prover exceeds the time limit.
- `OutOfMemory`: the prover exceeds the memory limit.
- `Unknown` *msg*: the prover can't determine if the task is valid; the string parameter *msg* indicates some extra information.
- `Failure` *msg*: the prover reports a failure, *e.g.* it was unable to read correctly its input task.

- `HighFailure`: an error occurred while trying to call the prover, or the prover answer was not understood (*e.g.* none of the given regular expressions in the driver file matches the output of the prover).

Here is thus another way of calling the Alt-Ergo prover, on our second task.

```
let result2 : Call_provers.prover_result =
  Call_provers.wait_on_call
    (Driver.prove_task ~command:alt_ergo.Whyconf.command
    ~limit:{Call_provers.empty_limit with Call_provers.limit_time = 10}
    alt_ergo_driver task2)

let () = printf "@[On task 2, alt-ergo answers %a in %5.2f seconds@."
  Call_provers.print_prover_answer result1.Call_provers.pr_answer
  result1.Call_provers.pr_time
```

The output of our program is now as follows.

```
On task 1, alt-ergo answers Valid (0.01s)
On task 2, alt-ergo answers Valid in  0.01 seconds
```

3.4 Building Terms

An important feature of the functions for building terms and formulas is that they statically guarantee that only well-typed terms can be constructed.

Here is the way we build the formula $2 + 2 = 4$. The main difficulty is to access the internal identifier for addition: it must be retrieved from the standard theory `Int` of the file `int.why`.

```
let two : Term.term = Term.t_nat_const 2
let four : Term.term = Term.t_nat_const 4
let int_theory : Theory.theory = Env.read_theory env ["int"] "Int"
let plus_symbol : Term.lsymbol =
  Theory.ns_find_ls int_theory.Theory.th_export ["infix +"]
let two_plus_two : Term.term = Term.t_app_infer plus_symbol [two;two]
let fmla3 : Term.term = Term.t_equ two_plus_two four
```

An important point to notice as that when building the application of + to the arguments, it is checked that the types are correct. Indeed the constructor `t_app_infer` infers the type of the resulting term. One could also provide the expected type as follows.

```
let two_plus_two : Term.term = Term.fs_app plus_symbol [two;two] Ty.ty_int
```

When building a task with this formula, we need to declare that we use theory `Int`:

```
let task3 = None
let task3 = Task.use_export task3 int_theory
let goal_id3 = Decl.create_prsymbol (Ident.id_fresh "goal3")
let task3 = Task.add_prop_decl task3 Decl.Pgoal goal_id3 fmla3
```

3.5 Building Quantified Formulas

To illustrate how to build quantified formulas, let us consider the formula $\forall x : int.x*x \geq 0$. The first step is to obtain the symbols from `Int`.

```
let zero : Term.term = Term.t_nat_const 0
let mult_symbol : Term.lsymbol =
  Theory.ns_find_ls int_theory.Theory.th_export ["infix *"]
let ge_symbol : Term.lsymbol =
  Theory.ns_find_ls int_theory.Theory.th_export ["infix >="]
```

The next step is to introduce the variable x with the type int.

```
let var_x : Term.vsymbol =
  Term.create_vsymbol (Ident.id_fresh "x") Ty.ty_int
```

The formula $x * x \geq 0$ is obtained as in the previous example.

```
let x : Term.term = Term.t_var var_x
let x_times_x : Term.term = Term.t_app_infer mult_symbol [x;x]
let fmla4_aux : Term.term = Term.ps_app ge_symbol [x_times_x;zero]
```

To quantify on x, we use the appropriate smart constructor as follows.

```
let fmla4 : Term.term = Term.t_forall_close [var_x] [] fmla4_aux
```

3.6 Building Theories

We illustrate now how one can build theories. Building a theory must be done by a sequence of calls:

- creating a theory "under construction", of type `Theory.theory_uc`;
- adding declarations, one at a time;
- closing the theory under construction, obtaining something of type `Theory.theory`.

Creation of a theory named `My_theory` is done by

```
let my_theory : Theory.theory_uc =
  Theory.create_theory (Ident.id_fresh "My_theory")
```

First let us add formula 1 above as a goal:

```
let decl_goal1 : Decl.decl =
  Decl.create_prop_decl Decl.Pgoal goal_id1 fmla1
let my_theory : Theory.theory_uc = Theory.add_decl my_theory decl_goal1
```

Note that we reused the goal identifier `goal_id1` that we already defined to create task 1 above.

Adding formula 2 needs to add the declarations of predicate variables A and B first:

```
let my_theory : Theory.theory_uc =
  Theory.add_param_decl my_theory prop_var_A
let my_theory : Theory.theory_uc =
  Theory.add_param_decl my_theory prop_var_B
```

```
let decl_goal2 : Decl.decl =
  Decl.create_prop_decl Decl.Pgoal goal_id2 fmla2
let my_theory : Theory.theory_uc = Theory.add_decl my_theory decl_goal2
```

Adding formula 3 is a bit more complex since it uses integers, thus it requires to "use" the theory `int.Int`. Using a theory is indeed not a primitive operation in the API: it must be done by a combination of an "export" and the creation of a namespace. We provide a helper function for that:

```
(* helper function: [use th1 th2] insert the equivalent of a
   "use import th2" in theory th1 under construction *)
let use th1 th2 =
  let name = th2.Theory.th_name in
  Theory.close_scope
    (Theory.use_export (Theory.open_scope th1 name.Ident.id_string) th2)
    ~import:true
```

Addition of formula 3 is then

```
let my_theory : Theory.theory_uc = use my_theory int_theory
let decl_goal3 : Decl.decl =
  Decl.create_prop_decl Decl.Pgoal goal_id3 fmla3
let my_theory : Theory.theory_uc = Theory.add_decl my_theory decl_goal3
```

Addition of goal 4 is nothing more complex:

```
let decl_goal4 : Decl.decl =
  Decl.create_prop_decl Decl.Pgoal goal_id4 fmla4
let my_theory : Theory.theory_uc = Theory.add_decl my_theory decl_goal4
```

Finally, we close our theory under construction as follows.

```
let my_theory : Theory.theory = Theory.close_theory my_theory
```

We can inspect what we did by printing that theory:

```
let () = printf "@[my new theory is as follows:@\n@\n%a@]@."
              Pretty.print_theory my_theory
```

which outputs

```
my new theory is as follows:

theory My_theory
  (* use BuiltIn *)

  goal goal1 : true \/ false

  predicate A

  predicate B

  goal goal2 : A /\ B -> A

  (* use int.Int *)
```

```
  goal goal3 : (2 + 2) = 4

  goal goal4 : forall x:int. (x * x) >= 0
end
```

From a theory, one can compute at once all the proof tasks it contains as follows:

```
let my_tasks : Task.task list =
  List.rev (Task.split_theory my_theory None None)
```

Note that the tasks are returned in reverse order, so we reverse the list above.

We can check our generated tasks by printing them:

```
let () =
  printf "Tasks are:@.";
  let _ =
    List.fold_left
      (fun i t -> printf "Task %d: %a@." i Pretty.print_task t; i+1)
      1 my_tasks
  in ()
```

One can run provers on those tasks exactly as we did above.

3.7 Operations on Terms and Formulas, Transformations

The following code illustrates a simple recursive functions of formulas. It explores the formula and when a negation is found, it tries to push it down below a conjunction, a disjunction or a quantifier.

```
open Term

let rec negate (t:term) : term =
  match t.t_node with
  | Ttrue -> t_false
  | Tfalse -> t_true
  | Tnot t -> t
  | Tbinop(Tand,t1,t2) -> t_or (negate t1) (negate t2)
  | Tbinop(Tor,t1,t2) -> t_and (negate t1) (negate t2)
  | Tquant(Tforall,tq) ->
     let vars,triggers,t' = t_open_quant tq in
     let tq' = t_close_quant vars triggers (negate t') in
     t_exists tq'
  | Tquant(Texists,tq) ->
     let vars,triggers,t' = t_open_quant tq in
     let tq' = t_close_quant vars triggers (negate t') in
     t_forall tq'
  | _ -> t_not t

let rec traverse (t:term) : term =
  match t.t_node with
  | Tnot t -> t_map traverse (negate t)
```

```
  | _ -> t_map traverse t
```

The following illustrates how to turn such an OCaml function into a transformation in the sense of the Why3 API. Moreover, it registers that transformation to make it available for example in Why3 IDE.

```
let negate_goal pr t = [Decl.create_prop_decl Decl.Pgoal pr (traverse t)]

let negate_trans = Trans.goal negate_goal

let () = Trans.register_transform
          "push_negations_down" negate_trans
          ~desc:"In the current goal,@ push negations down,@ \
                 across logical connectives."
```

The directory `src/transform` contains the code for the many transformations that are already available in Why3.

3.8 Proof Sessions

See the example `examples/use_api/create_session.ml` of the distribution for an illustration on how to manipulate proof sessions from an OCaml program.

3.9 ML Programs

The simplest way to build WhyML programs from OCaml is to build untyped syntax trees for such programs, and then call the Why3 typing procedure to build typed declarations.

The examples of this section are available in the file `examples/use_api/mlw_tree.ml` of the distribution.

The first step is to build an environment as already illustrated in Section 3.3, and open the OCaml module `Ptree` which contains most of the OCaml functions we need in this section.

```
open Why3
let config : Whyconf.config = Whyconf.read_config None
let main : Whyconf.main = Whyconf.get_main config
let env : Env.env = Env.create_env (Whyconf.loadpath main)
open Ptree
```

To contain all the example programs we are going to build we need a module. We start the creation of that module using the following declarations, that first introduces a pseudo "file" to hold the module, then the module itself called `Program`.

```
let () = Typing.open_file env [] (* empty pathname *)
let mk_ident s = { id_str = s; id_ats = []; id_loc = Loc.dummy_position }
let () = Typing.open_module (mk_ident "Program")
```

Notice the use of a first simple helper function `mk_ident` to build an identifier without any attributes nor any location.

To write our programs, we need to import some other modules from the standard library. The following introduces two helper functions for building qualified identifiers and importing modules, and finally imports `int.Int`.

```
let mk_qid l =
  let rec aux l =
    match l with
      | [] -> assert false
      | [x] -> Qident(mk_ident x)
      | x::r -> Qdot(aux r,mk_ident x)
  in
  aux (List.rev l)

let use_import (f, m) =
  let m = mk_ident m in
  let loc = Loc.dummy_position in
  Typing.open_scope loc m;
  Typing.add_decl loc (Ptree.Duse (Qdot (Qident (mk_ident f), m)));
  Typing.close_scope loc ~import:true

let use_int_Int = use_import ("int","Int")
```

We want now to build a program equivalent to the following code in concrete Why3 syntax.

```
    let f1 (x:int) : unit
       requires { x=6 }
       ensures { result=42 }
     = x*7
```

The OCaml code that programmatically build this Why3 function is as follows.

```
let eq_symb = mk_qid [Ident.op_infix "="]
let int_type_id = mk_qid ["int"]
let int_type = PTtyapp(int_type_id,[])
let mul_int = mk_qid ["Int";Ident.op_infix "*"]

let d1 : decl =
  let id_x = mk_ident "x" in
  let pre = mk_tapp eq_symb [mk_var id_x; mk_tconst "6"] in
  let result = mk_ident "result" in
  let post = mk_tapp eq_symb [mk_var result; mk_tconst "42"] in
  let spec = {
    sp_pre = [pre];
    sp_post = [Loc.dummy_position,[pat_var result,post]];
    sp_xpost = [];
    sp_reads = [];
    sp_writes = [];
    sp_alias = [];
    sp_variant = [];
    sp_checkrw = false;
    sp_diverge = false;
    sp_partial = false;
  }
  in
```

```
let mk_expr e = { expr_desc = e; expr_loc = Loc.dummy_position }

let mk_term t = { term_desc = t; term_loc = Loc.dummy_position }

let mk_pat p = { pat_desc = p; pat_loc = Loc.dummy_position }
let pat_var id = mk_pat (Pvar id)

let mk_var id = mk_term (Tident (Qident id))

let param0 = [Loc.dummy_position, None, false, Some (PTtuple [])]
let param1 id ty = [Loc.dummy_position, Some id, false, Some ty]

let mk_tconst s =
  mk_term
    (Tconst
       Number.(ConstInt { ic_negative = false ;
                          ic_abs = int_literal_dec s }))

let mk_econst s =
  mk_expr
    (Econst
       Number.(ConstInt { ic_negative = false ;
                          ic_abs = int_literal_dec s }))

let mk_tapp f l = mk_term (Tidapp(f,l))

let mk_eapp f l = mk_expr (Eidapp(f,l))

let mk_evar x = mk_expr(Eident(Qident x))
```

Figure 3.1: Helper functions for building WhyML programs

```
  let body = mk_eapp mul_int [mk_evar id_x; mk_econst "7"] in
  let f1 =
    Efun(param1 id_x int_type, None, Ity.MaskVisible, spec, body)
  in
  Dlet(mk_ident "f1",false,Expr.RKnone, mk_expr f1)

let () =
  try Typing.add_decl Loc.dummy_position d1
  with e ->
    Format.printf "Exception raised during typing of d:@ %a@."
      Exn_printer.exn_printer e
```

This code makes uses of helper functions that are given in Figure 3.1.

We want now to build a program equivalent to the following code in concrete Why3 syntax.

```
    let f2 () : int
       requires { true }
```

```
        ensures { result >= 0 }
      = let x = ref 42 in !x
```

We need to import the `ref.Ref` module first. The rest is similar to the first example, the code is as follows

```
let ge_int = mk_qid ["Int";Ident.op_infix ">="]

let use_ref_Ref = use_import ("ref","Ref")

let d2 =
  let result = mk_ident "result" in
  let post = mk_term(Tidapp(ge_int,[mk_var result;mk_tconst "0"])) in
  let spec = {
    sp_pre = [];
    sp_post = [Loc.dummy_position,[pat_var result,post]];
    sp_xpost = [];
    sp_reads = [];
    sp_writes = [];
    sp_alias = [];
    sp_variant = [];
    sp_checkrw = false;
    sp_diverge = false;
    sp_partial = false;
  }
  in
  let body =
    let e1 = mk_eapp (mk_qid ["Ref";"ref"]) [mk_econst "42"] in
    let id_x = mk_ident "x" in
    let e2 = mk_eapp (mk_qid ["Ref";Ident.op_prefix "!"]) [mk_evar id_x] in
    mk_expr(Elet(id_x,false,Expr.RKlocal,e1,e2))
  in
  let f = Efun(param0,None,Ity.MaskVisible,spec,body)
  in
  Dlet(mk_ident "f2",false,Expr.RKnone, mk_expr f)

let () =
  try Typing.add_decl Loc.dummy_position d2
  with e ->
    Format.printf "Exception raised during typing of d2:@ %a@."
      Exn_printer.exn_printer e
```

The next example makes use of arrays.

```
let f (a:array int) : unit
      requires { a.length >= 1 }
      ensures { a[0] = 42 }
    = a[0] <- 42
```

The corresponding OCaml code is as follows

```
let () = use_import ("array","Array")
```

```
let array_int_type = PTtyapp(mk_qid ["Array";"array"],[int_type])

let length = mk_qid ["Array";"length"]

let array_get = mk_qid ["Array"; Ident.op_get ""]

let array_set = mk_qid ["Array"; Ident.op_set ""]

let d3 =
  let id_a = mk_ident "a" in
  let pre =
    mk_tapp ge_int [mk_tapp length [mk_var id_a]; mk_tconst "1"]
  in
  let post =
    mk_tapp eq_symb [mk_tapp array_get [mk_var id_a; mk_tconst "0"];
                     mk_tconst "42"]
  in
  let spec = {
    sp_pre = [pre];
    sp_post = [Loc.dummy_position,[mk_pat Pwild,post]];
    sp_xpost = [];
    sp_reads = [];
    sp_writes = [];
    sp_alias = [];
    sp_variant = [];
    sp_checkrw = false;
    sp_diverge = false;
    sp_partial = false;
  }
  in
  let body =
    mk_eapp array_set [mk_evar id_a; mk_econst "0"; mk_econst "42"]
  in
  let f = Efun(param1 id_a array_int_type,
               None,Ity.MaskVisible,spec,body)
  in
  Dlet(mk_ident "f3", false, Expr.RKnone, mk_expr f)

let () =
  try Typing.add_decl Loc.dummy_position d3
  with e ->
    Format.printf "Exception raised during typing of d3:@ %a@."
      Exn_printer.exn_printer e
```

Having declared all the programs we wanted to write, we can now close the module and the file, and get as a result the set of modules of our file, under the form of a map of module names to modules.

```
let () = Typing.close_module Loc.dummy_position
let mods : Pmodule.pmodule Wstdlib.Mstr.t = Typing.close_file ()
```

We can then construct the proofs tasks for our module, and then try to call the Alt-Ergo prover. The rest of that code is using OCaml functions that were already introduced before.

```
let my_tasks : Task.task list =
  let mods =
    Wstdlib.Mstr.fold
      (fun _ m acc ->
       List.rev_append
         (Task.split_theory m.Pmodule.mod_theory None None) acc)
      mods []
  in List.rev mods

open Format

let () =
  printf "Tasks are:@.";
  let _ =
    List.fold_left
      (fun i t -> printf "Task %d: %a@." i Pretty.print_task t; i+1)
      1 my_tasks
  in ()

let provers : Whyconf.config_prover Whyconf.Mprover.t =
  Whyconf.get_provers config

let alt_ergo : Whyconf.config_prover =
  let fp = Whyconf.parse_filter_prover "Alt-Ergo" in
  (** all provers that have the name "Alt-Ergo" *)
  let provers = Whyconf.filter_provers config fp in
  if Whyconf.Mprover.is_empty provers then begin
    eprintf "Prover Alt-Ergo not installed or not configured@.";
    exit 0
  end else
    snd (Whyconf.Mprover.max_binding provers)

let alt_ergo_driver : Driver.driver =
  try
    Whyconf.load_driver main env alt_ergo.Whyconf.driver []
  with e ->
    eprintf "Failed to load driver for alt-ergo: %a@."
      Exn_printer.exn_printer e;
    exit 1

let () =
  let _ =
    List.fold_left
      (fun i t ->
       let r =
         Call_provers.wait_on_call
```

```
        (Driver.prove_task ~limit:Call_provers.empty_limit
                           ~command:alt_ergo.Whyconf.command
                           alt_ergo_driver t)
    in
    printf "@[On task %d, alt-ergo answers %a@."
           i Call_provers.print_prover_result r;
    i+1
   )
   1 my_tasks
in ()
```

3.10 Generating counterexamples

That feature is presented in details in Section 5.3.7, that should be read first. The counterexamples can also be generated using the API. The following explains how to change the source code (mainly adding attributes) in order to display counterexamples and how to parse the result given by Why3. To illustrate this, we will adapt the examples from Section 3.1 to display counterexamples.

3.10.1 Attributes and locations on identifiers

For variables to be used for counterexamples they need to contain an attribute called `model_trace` and a location. The `model_trace` states the name the user wants the variable to be named in the output of the counterexamples pass. Usually, people put a reference to their program AST node in this attribute: this helps them to parse and display the results given by Why3. The locations are also necessary as every counterexamples values with no location won't be displayed. For example, an assignment of the source language such as the following will probably trigger the creation of an ident (for the left value) in a user subsequent tasks:

```
x := !y + 1
```

This means that the ident generated for x will hold both a `model_trace` and a location.

The example becomes the following:

```
let make_attribute (name: string) : Ident.attribute =
  Ident.create_attribute ("model_trace:" ^ name)
let prop_var_A : Term.lsymbol =
  (* [user_position file line left_col right_col] *)
  let loc = Loc.user_position "myfile.my_ext" 28 0 0 in
  let attrs = Ident.Sattr.singleton (make_attribute "my_A") in
  Term.create_psymbol (Ident.id_fresh ~attrs ~loc "A") []
```

In the above, we defined a proposition ident with a location and a `model_trace`.

3.10.2 Attributes in formulas

Now that variables are tagged, we can define formulas. To define a goal formula for counterexamples, we need to tag it with the `vc:annotation` attribute. This attribute is automatically added when using the VC generation of Why3, but on a user-built task, this needs to be added. We also need to add a location for this goal. The following is obtained for the simple formula linking A and B:

```
let atom_A : Term.term = Term.ps_app prop_var_A []
let atom_B : Term.term = Term.ps_app prop_var_B []
(* Voluntarily wrong verification condition *)
let fmla2 : Term.term =
  Term.t_implies atom_A (Term.t_and atom_A atom_B)
(* We add a location and attribute to indicate the start of a goal *)
let fmla2 : Term.term =
  let loc = Loc.user_position "myfile.my_ext" 42 28 91 in
  let attrs = Ident.Sattr.singleton Ity.annot_attr in
  (* Note that this remove any existing attribute/locations on fmla2 *)
  Term.t_attr_set ~loc attrs fmla2
```

Note: the transformations used for counterexamples will create new variables for each variable occuring inside the formula tagged by `vc:annotation`. These variables are duplicates located at the VC line. They allow giving all counterexample values located at that VC line.

3.10.3 Counterexamples output formats

Several output formats are available for counterexamples. For users who want to pretty-print their counterexamples values, we recommend to use the JSON output as follows:

```
(* prints Cvc4 answer *)
let () = printf "@[On task 1, Cvc4 answers %a@."
  Call_provers.print_prover_result result1

let () = printf "Model is %a@."
    (Model_parser.print_model_json ?me_name_trans:None ?vc_line_trans:None)
    result1.Call_provers.pr_model
```

Part II

Reference Manual

Chapter 4

Compilation, Installation

4.1 Installing Why3

4.1.1 Installation via OPAM

The simplest way to install Why3 is via OPAM, the OCaml package manager. It is as simple as

```
> opam install why3
```

Then jump to Section 4.2 to install external provers.

4.1.2 Installation Instructions from Source Distribution

In short, installation from sources proceeds as follows.

```
> ./configure
> make
> make install (as super-user)
```

After unpacking the distribution, go to the newly created directory `why3-1.2.1`. Compilation must start with a configuration phase which is run as

```
> ./configure
```

This analyzes your current configuration and checks if requirements hold. Compilation requires:

- The Objective Caml compiler. It is available as a binary package for most Unix distributions. For Debian-based Linux distributions, you can install the packages

  ```
  ocaml ocaml-native-compilers
  ```

  It is also installable from sources, downloadable from the site http://caml.inria.fr/ocaml/

For some of the Why3 tools, additional OCaml libraries are needed:

- For the graphical interface, the Lablgtk2 library is needed. It provides OCaml bindings of the gtk2 graphical library. For Debian-based Linux distributions, you can install the packages

`liblablgtk2-ocaml-dev liblablgtksourceview2-ocaml-dev`

It is also installable from sources, available from the site http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html

If you want to use the Coq realizations (Section 8.2), then Coq has to be installed before Why3. Look at the summary printed at the end of the configuration script to check if Coq has been detected properly. Similarly, in order to use PVS (Section 8.5) or Isabelle (Section 8.4) to discharge proofs, PVS and Isabelle must be installed before Why3. You should check that those proof assistants are correctly detected by the configure script.

When configuration is finished, you can compile Why3.

```
make
```

Installation is performed (as super-user if needed) using

```
make install
```

Installation can be tested as follows:

1. install some external provers (see Section 4.2 below)
2. run `why3 config --detect`
3. run some examples from the distribution, *e.g.* you should obtain the following (provided the required provers are installed on your machine):

```
$ cd examples
$ why3 replay logic/scottish-private-club
 1/1 (replay OK)
$ why3 replay same_fringe
 18/18 (replay OK)
```

Local Use, Without Installation

It is not mandatory to install Why3 into system directories. Why3 can be configured and compiled for local use as follows:

```
./configure --enable-local
make
```

The Why3 executables are then available in the subdirectory `bin/`. This directory can be added in your `PATH`.

Installation of the Why3 API

By default, the Why3 API is not installed. It can be installed using

```
make byte opt
make install-lib (as super-user)
```

4.2 Installing External Provers

Why3 can use a wide range of external theorem provers. These need to be installed separately, and then Why3 needs to be configured to use them. There is no need to install automatic provers, *e.g.* SMT solvers, before compiling and installing Why3. For installation of external provers, please refer to the specific section about provers from http://why3.lri.fr/. (If you have installed Why3 via OPAM, note that you can install the SMT solver Alt-Ergo via OPAM as well.)

Once you have installed a prover, or a new version of a prover, you have to run the following command:

```
> why3 config --detect
```

It scans your `PATH` for provers and updates your configuration file (see Section 5.1) accordingly.

4.2.1 Multiple Versions of the Same Prover

Why3 is able to use several versions of the same prover, *e.g.* it can use both CVC4 1.4 and CVC4 1.5 at the same time. The automatic detection of provers looks for typical names for their executable command, *e.g.* `cvc4` for CVC3. However, if you install several versions of the same prover it is likely that you would use specialized executable names, such as `cvc4-1.4` or `cvc4-1.5`. If needed, option `--add-prover` can be added to the `config` command to specify names of prover executables, *e.g.*

```
why3 config --add-prover cvc4 cvc4-dev /usr/local/bin/cvc4-dev
```

the first argument (here `cvc4`) must be one of the family of provers known. The list of these famillies can be obtain using

```
why3 config --list-prover-families
```

as they are in fact listed in the file `provers-detection-data.conf`, typically located in `/usr/local/share/why3` after installation. See Appendix 9.2 for details.

4.2.2 Session Update after Prover Upgrade

If you happen to upgrade a prover, *e.g.* installing CVC4 1.5 in place of CVC4 1.4, then the proof sessions formerly recorded will still refer to the old version of the prover. If you open one such a session with the GUI, and replay the proofs, a popup window will show up for asking you to choose between three options:

- Keep the former proof attempts as they are, with the old prover version. They will not be replayed.
- Remove the former proof attempts.
- Upgrade the former proof attempts to an installed prover (typically an upgraded version). The corresponding proof attempts will become attached to this new prover, and marked as obsolete, to make their replay mandatory. If a proof attempt with this installed prover is already present the old proof attempt is just removed. Note that you need to invoke again the replay command to replay those proof attempts.

- Copy the former proofs to an installed prover. This is a combination of the actions above: each proof attempt is duplicated, one with the former prover version, and one for the new version marked as obsolete.

Notice that if the prover under consideration is an interactive one, then the copy option will duplicate also the edited proof scripts, whereas the upgrade-without-copy option will just reuse the former proof scripts.

Your choice between the three options above will be recorded, one for each prover, in the Why3 configuration file. Within the GUI, you can discard these choices via the Preferences dialog: just click on one choice to remove it.

Outside the GUI, the prover upgrades are handled as follows. The `replay` command will take into account any prover upgrade policy stored in the configuration. The `session` command performs move or copy operations on proof attempts in a fine-grained way, using filters, as detailed in Section 5.5.

Chapter 5

Reference Manuals for the Why3 Tools

This chapter details the usage of each of the command-line tools provided by the Why3 environment. The main command is `why3`; it acts as an entry-point to all the features of Why3. It is invoked as such

```
why3 [general options...] <command> [specific options...]
```

The following commands are available:

`config` manages the user's configuration, including the detection of installed provers.

`doc` produces HTML versions of Why3 source codes.

`execute` performs a symbolic execution of WhyML input files.

`extract` generates an OCaml program corresponding to WhyML input files.

`ide` provides a graphical interface to display goals and to run provers and transformations on them.

`prove` reads WhyML input files and calls provers, on the command-line.

`realize` generates interactive proof skeletons for Why3 input files.

`replay` replays the proofs stored in a session, for regression test purposes.

`session` dumps various informations from a proof session, and possibly modifies the session.

`wc` gives some token statistics about WhyML source files.

All these commands are also available as standalone executable files, if needed.

The commands accept a common subset of command-line options. In particular, option `--help` displays the usage and options.

`-L <dir>` adds *`<dir>`* in the load path, to search for theories.

`--library <dir>` is the same as `-L`.

`-C <file>` reads the configuration from the given file.

--config <*file*> is the same as **-C**.

--extra-config <*file*> reads additional configuration from the given file.

--list-debug-flags list known debug flags.

--list-transforms list known transformations.

--list-printers list known printers.

--list-provers list known provers

--list-formats list known input formats

--list-metas list known metas

--debug-all sets all debug flags (except flags which change the behavior).

--debug <*flag*> sets a specific debug flag.

--help displays the usage and the exact list of options for the given tool.

5.1 The `config` Command

Why3 must be configured to access external provers. Typically, this is done by running the `config` command. This must be done each time a new prover is installed.

The provers that Why3 attempts to detect are described in the readable configuration file `provers-detection-data.conf` of the Why3 data directory (*e.g.* `/usr/local/share/why3`). Advanced users may try to modify this file to add support for detection of other provers. (In that case, please consider submitting a new prover configuration on the bug tracking system.)

The result of provers detection is stored in the user's configuration file (`~/.why3.conf` or, in the case of local installation, `why3.conf` in Why3 sources top directory). This file is also human-readable, and advanced users may modify it in order to experiment with different ways of calling provers, *e.g.* different versions of the same prover, or with different options.

The `config` command also detects the plugins installed in the Why3 plugins directory (*e.g.* `/usr/local/lib/why3/plugins`). A plugin must register itself as a parser, a transformation or a printer, as explained in the corresponding section.

If the user's configuration file is already present, `config` will only reset unset variables to default value, but will not try to detect provers. The option `--detect-provers` should be used to force Why3 to detect again the available provers and to replace them in the configuration file. The option `--detect-plugins` will do the same for plugins.

If a supported prover is installed under a name that is not automatically recognized by `why3config`, the option `--add-prover` will add a specified binary to the configuration. For example, an Alt-Ergo executable `/home/me/bin/alt-ergo-trunk` can be added as follows:

```
why3 config --add-prover alt-ergo alt-ergo-trunk /home/me/bin/alt-ergo-trunk
```

As the first argument, one should put a prover family identification string. The list of known prover families can be obtained by the option `--list-prover-families`.

5.2 The prove Command

Why3 is primarily used to call provers on goals contained in an input file. By default, such a file must be written in WhyML language (extension `.mlw`). However, a dynamically loaded plugin can register a parser for some other format of logical problems, *e.g.* TPTP or SMT-LIB.

The `prove` command executes the following steps:

1. Parse the command line and report errors if needed.
2. Read the configuration file using the priority defined in Section 9.3.
3. Load the plugins mentioned in the configuration. It will not stop if some plugin fails to load.
4. Parse and typecheck the given files using the correct parser in order to obtain a set of Why3 theories for each file. It uses the filename extension or the `--format` option to choose among the available parsers. `why3 --list-formats` lists the registered parsers. WhyML modules are turned into theories containing verification conditions as goals.
5. Extract the selected goals inside each of the selected theories into tasks. The goals and theories are selected using options `-G/--goal` and `-T/--theory`. Option `-T/--theory` applies to the previous file appearing on the command line. Option `-G/--goal` applies to the previous theory appearing on the command line. If no theories are selected in a file, then every theory is considered as selected. If no goals are selected in a theory, then every goal is considered as selected.
6. Apply the transformations requested with `-a/--apply-transform` in their order of appearance on the command line. `why3 --list-transforms` lists the known transformations; plugins can add more of them.
7. Apply the driver selected with the `-D/--driver` option, or the driver of the prover selected with the `-P/--prover` option. `why3 --list-provers` lists the known provers, *i.e.* the ones that appear in the configuration file.
8. If option `-P/--prover` is given, call the selected prover on each generated task and print the results. If option `-D/--driver` is given, print each generated task using the format specified in the selected driver.

5.2.1 Prover Results

The provers can give the following output:

Valid The goal is proved in the given context.

Unknown The prover has stopped its search.

Timeout The prover has reached the time limit.

Failure An error has occurred.

Invalid The prover knows the goal cannot be proved.

5.2.2 Additional Options

`--extra-expl-prefix <s>` specifies *s* as an additional prefix for labels that denotes VC explanations. The option can be used several times to specify several prefixes.

5.2.3 Getting Potential Counterexamples

That feature is presented in details in Section 5.3.7, that should be read first.

Counterexamples are also displayed by the `why3 prove` command when one selects a prover with the `counterexamples` alternative. The output is currently done in a JSON syntax (this may change in the future).

5.3 The `ide` Command

The basic usage of the GUI is described by the tutorial of Section 1.2. The command-line options are the common options detailed in introduction to this chapter, plus the specific option already described for the command `prove` in Section 5.2.2.

`--extra-expl-prefix <s>`

At least one anonymous argument must be specified on the command line. More precisely, the first anonymous argument must be the directory of the session. If the directory does not exist, it is created. The other arguments should be existing files that are going to be added to the session. For convenience, if there is only one anonymous argument, it can be an existing file and in this case the session directory is obtained by removing the extension from the file name.

We describe the actions of the various menus and buttons of the interface.

5.3.1 Session

Why3 stores in a session the way you achieve to prove goals that come from a file (`.why`), from weakest-precondition (`.mlw`) or by other means. A session stores which file you prove, by applying which transformations, by using which prover. A proof attempt records the complete name of a prover (name, version, optional attribute), the time limit and memory limit given, and the result of the prover. The result of the prover is the same as when you run the `prove` command. It contains the time taken and the state of the proof:

Valid The task is valid according to the prover. The goal is considered proved.

Invalid The task is invalid.

Timeout the prover exceeded the time limit.

OufOfMemory The prover exceeded the memory limit.

Unknown The prover cannot determine if the task is valid. Some additional information can be provided.

Failure The prover reported a failure.

HighFailure An error occurred while trying to call the prover, or the prover answer was not understood.

Additionally, a proof attempt can have the following attributes:

obsolete The prover associated to that proof attempt has not been run on the current task, but on an earlier version of that task. You need to replay the proof attempt, *i.e.* run the prover with the current task of the proof attempt, in order to update the answer of the prover and remove this attribute.

detached The proof attempt is not associated to a proof task anymore. The reason might be that a proof goal disappeared, or that there is a syntax or typing error in the current file, that makes all nodes temporarily detached until the parsing error is fixed. Detached nodes of the session tree are kept until they are explicitly removed, either using a remove command or the clean command. They can be reused, as any other nodes, using the copy/paste operation.

Generally, proof attempts are marked obsolete just after the start of the user interface. Indeed, when you load a session in order to modify it (not with `why3session info` for instance), Why3 rebuilds the goals to prove by using the information provided in the session. If you modify the original file (`.mlw`) or if the transformations have changed (new version of Why3), Why3 will detect that. Since the provers might answer differently on these new proof obligations, the corresponding proof attempts are marked obsolete.

5.3.2 Context Menu

The left toolbar that was present in former versions of Why3 is now replaced by a context menu activited by clicking the right mouse button, while cursor is on a given row of the proof session tree.

provers The detected provers are listed. Note that you can hide some provers of that list using the preferences, tab Provers.

strategies the set of known strategies is listed

Edit starts an editor on the selected task.

Replay valid obsolete proofs all proof nodes below the selected nodes that are obsolete but whose former status was Valid are replayed.

Replay all obsolete proofs all proof nodes below the selected nodes that are obsolete are replayed.

Remove removes a proof attempt or a transformation.

Clean removes any unsuccessful proof attempt for which there is another successful proof attempt for the same goal

Interrupt cancels all the proof attempts currently scheduled or running.

5.3.3 Global Menus

Menu File

Add File to session adds a file in the current proof session.

Preferences opens a window for modifying preferred configuration parameters, see details below.

Save session saves current session state on disk. The policy to decide when to save the session is configurable, as described in the preferences below.

Save files saves edited soruce files on disk.

Save session and files saves both current session state and edited files on disk.

Save all and Refresh session save session and edited files, and refresh the current session tree.

Quit exits the GUI.

Menu Tools

Strategies section provides a set of actions that are performed on the selected goal(s):

Split VC splits the current goal into subgoals.

Auto level 0 is a basic proof search strategy that applies a few provers on the goal with a short time limit.

Auto level 1 is a strategy that first applies a few provers on the goal with a short time limit, then splits the goal and tries again on the subgoals

Auto level 2 is a strategy more elaborate than level 1, that attempts to apply a few transformations that are typically useful. It also tries the provers with a larger time limit.

A more detailed description of strategies is given in Section 9.6, as well as a description on how to design strategies of your own.

Provers provide a menu item for each detected prover. Clicking on such an item starts the corresponding prover on the selected goal(s). To start a prover with a different time limit, you may either change the default time limit in the Preferences, or using the text command field and type the prover name followed by the time limit.

Transformations gives access to all the known transformations.

Edit starts an editor on the selected task.

For automatic provers, this allows to see the file sent to the prover.

For interactive provers, this also allows to add or modify the corresponding proof script. The modifications are saved, and can be retrieved later even if the goal was modified.

Replay valid obsolete proofs replays all the obsolete proofs below the current node whose former state was Valid.

Replay all obsolete proofs replays all the obsolete proofs below the current node.

Clean removes any unsuccessful proof attempt for which there is another successful proof attempt for the same goal

Remove removes a proof attempt or a transformation.

Mark obsolete marks all the proof as obsolete. This allows to replay every proof.

Interrupt cancels all the proof attempts currently scheduled or running.

Bisect performs a reduction of the context for the the current selected proof attempt, which must be a Valid one.

Focus focus the tree session view to the current node

Unfocus undoes the Focus action

Copy Marks of proof sub-tree for copy/past action

Paste Paste the previously selected sub-tree under the current node

Menu View

Enlarge font selects a large font

Reduce font selects a smaller font

Collapse proved goals closes all the rows of the tree view that are proved.

Expand All expands all the rows of the tree view.

Collapse under node closes all the rows of the tree view under the given node that are proved.

Expand below node expands the children below the current node

Expand all below node expands the whole subtree of the current node

Go to parent node move to the parent of the current node

Go to first child mode to the first child of the current node

Select next unproven goal go to the next unproven goal after the current node

Menu Help

Legend Explanations of the meaning of the various icons

About some information about this software.

5.3.4 Command-line interface

Between the top-right zone containing source files and task, and the bottom-right zone containing various messages, a text input field allows the user to invoke commands using a textual interface (see Figure 1.1). The 'help' command displays a basic list of available commands. All commands available in the menus are also available as a textual command. However the textual interface allows for much more possibilities, including the ability to invoke transformations with arguments.

5.3.5 Key shortcuts

- Save session and files : ctrl+s
- Save all and refresh session: ctrl+r
- Quit : ctrl+q
- Enlarge font : ctrl+plus
- Reduce font : ctrl+minus
- Collapse proven goals : !
- Collapse current node : -
- Expand current node : +
- Copy : ctrl+c

- Paste : ctrl+v
- Select parent node : ctrl+up
- Select next unproven goal : ctrl+down
- Change focus to command line : return
- Edit : e
- Replay : r
- Clean : c
- Remove : del
- Mark obsolete : o

5.3.6 Preferences Dialog

The preferences dialog allows you to customize various settings. They are grouped together under several tabs.

Note that there are to different buttons to close that dialog. The "Close" button will make modifications of any of these settings effective only for the current run of the GUI. The "Save&Close" button will save the modified settings in Why3 configuration file, to make them permanent.

General Settings tab allows one to set various general settings.

- the limits set on resource usages:
  - the time limit given to provers, in seconds
  - the memory given to provers, in megabytes
  - the maximal number of simultaneous provers allowed to run in parallel
- option to disallow source editing within the GUI
- the policy for saving sessions:
  - always save on exit (default): the current state of the proof session is saving on exit
  - never save on exit: the current state of the session is never saved automatically, you must use menu File/Save session
  - ask whether to save: on exit, a popup window asks whether you want to save or not.

Appearance settings tab

- show full task context: by default, only the local context of formulas is shown, that is only the declarations comming from the same module
- show attributes in formulas
- show corecions in formulas
- show source locations in formulas
- show time and memory limits for each proof

Finally, it is possible to choose an alternative icon set, provided, one is installed first.

Editors tab allows one to customize the use of external editors for proof scripts.

- The default editor to use when the Edit button is pressed.
- For each installed prover, a specific editor can be selected to override the default. Typically if you install the Coq prover, then the editor to use will be set to "CoqIDE" by default, and this dialog allows you to select the Emacs editor and its Proof General mode instead (`http://proofgeneral.inf.ed.ac.uk/`).

Provers tab allows to select which of the installed provers one wants to see in the context menu.

Uninstalled Provers tab presents all the decision previously taken for missing provers, as described in Section 4.2.2. You can remove any recorded decision by clicking on it.

5.3.7 Displaying Counterexamples

Why3 provides some support for extracting a potential counterexample from failing proof attempts, for provers that are able to produce a *counter-model* of the proof task. Why3 attempts to turn this counter-model into values for the free variables of the original Why3 input. Currently, this is supported for CVC4 prover version at least 1.5, and Z3 prover version at least 4.4.0.

The generation of counterexamples is fully integrated in Why3 IDE. The recommended usage is to first start a prover normally, as shown in Figure 5.1) and then click on the status icon for the corresponding proof attempt in the tree. Alternatively, one can use the key shortcut "G" or type `get-ce` in the command entry. The result can be seen on Figure 5.2: the same prover but with the alternative *counterexamples* is run. The resulting counterexample is displayed in two different ways. First, it is displayed in the Task tab of the top-right window, at the end of the text of the task, under the form of a list of pairs "variable = value", ordered by the line number of the source code in which that variable takes that value. Second, it is displayed in the Counterexample tab of the bottom right window, this time interleaved with the code, as shown in Figure 5.2.

Notes on format of displayed values

The counterexamples can contain values of various types.

- Integer or real variables are displayed in decimal.
- Bitvectors are displayed in hexadecimal
- Integer range types are displayed in a specific notation showing their projection to integers
- Floating-point numbers are displayed both under a decimal approximation and an exact hexadecimal value. The special values `+oo`, `-oo` and `NaN` may occur too.
- Values from algebraic types and record types are displayed as in the Why3 syntax
- Map values are displayed in a specific syntax detailed below

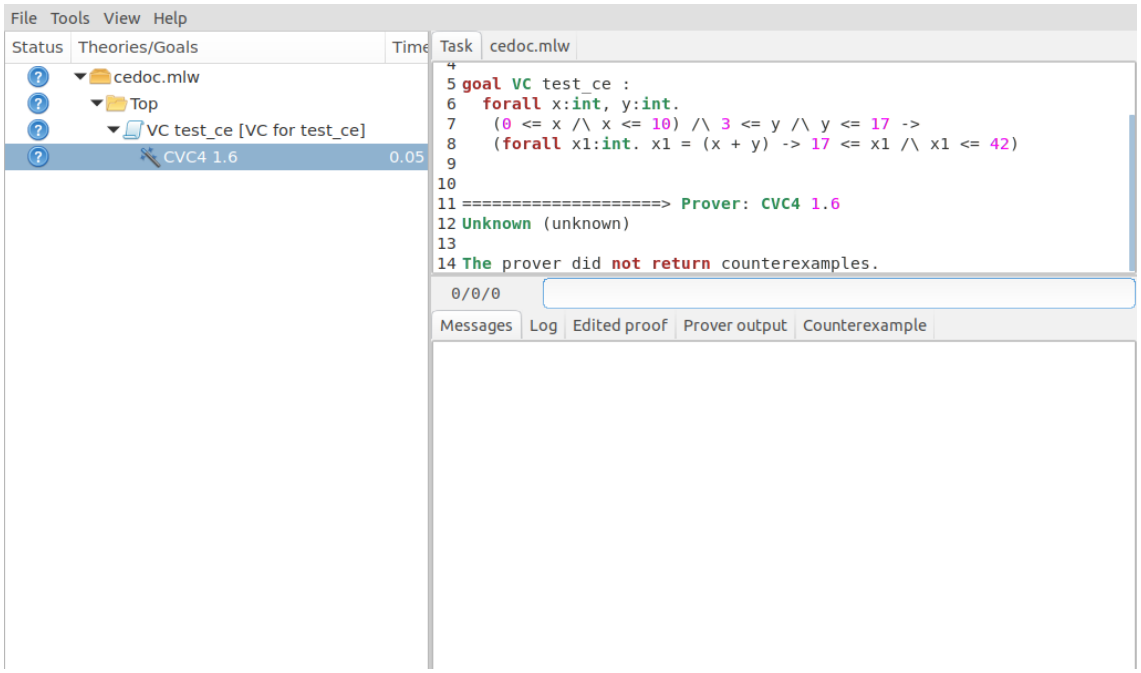

Figure 5.1: Failing execution of CVC4

Figure 5.2: Counterexamples display for CVC4

To detail the display of map values, consider the following code with a trivially false postcondition:

```
use int.Int
use ref.Ref
use map.Map

let ghost test_map (ghost x : ref (map int int)) : unit
  ensures { !x[0] <> !x[1] }
=
  x := Map.set !x 0 3
```

Executing CVC4 with the "counterexamples" alternative on goal will trigger counterexamples:

```
use int.Int
use ref.Ref
use map.Map

let ghost test_map (ghost x : ref (map int int)) : unit
(* x = (1 => 3,others => 0) *)
  ensures { !x[0] <> !x[1] }
  (* x = (0 => 3,1 => 3,others => 0) *)
=
  x := Map.set !x 0 3
  (* x = (0 => 3,1 => 3,others => 0) *)
```

The notation for map is to be understood with indices on left of the arrows and values on the right "(index => value)". The meaning of the keyword `others` is the value for all indices that were not mentioned yet. This shows that setting the parameter `x` to a map that has value 3 for index 1 and zero for all other indices is a counterexample. We can check that this negates the Why3ensures clause.

Known limitations

The counterexamples are known not to work on the following non-exhaustive list (which is undergoing active development):

- Code containing type polymorphism is often a problem due to the bad interaction between monomorphisation techniques and counterexamples. This is current an issue in particular for the Array module of the standard library.
- [TODO: complete this list]

More information on the implementation of counterexamples in Why3 can be found in [7] and in [5]. For the producing counterexamples using the Why3 API, see Section 3.10.

5.4 The `replay` Command

The `replay` command is meant to execute the proofs stored in a Why3 session file, as produced by the IDE. Its main purpose is to play non-regression tests. For instance, `examples/regtests.sh` is a script that runs regression tests on all the examples.

The tool is invoked in a terminal or a script using

`why3 replay` *`[options]`* `<project directory>`

The session file `why3session.xml` stored in the given directory is loaded and all the proofs it contains are rerun. Then, all the differences between the information stored in the session file and the new run are shown.

Nothing is shown when there is no change in the results, whether the considered goal is proved or not. When all the proof are done, a summary of what is proved or not is displayed using a tree-shape pretty print, similar to the IDE tree view after doing "Collapse proved goals". In other words, when a goal, a theory, or a file is fully proved, the subtree is not shown.

Obsolete proofs When some proof attempts stored in the session file are obsolete, the replay is run anyway, as with the replay button in the IDE. Then, the session file will be updated if both

- all the replayed proof attempts give the same result as what is stored in the session
- every goals are proved.

In other cases, you can use the IDE to update the session, or use the option `--force` described below.

Exit code and options The exit code is 0 if no difference was detected, 1 if there was. Other exit codes mean some failure in running the replay.

Options are:

`-s` suppresses the output of the final tree view.

`-q` runs quietly (no progress info).

`--force` enforces saving the session, if all proof attempts replayed correctly, even if some goals are not proved.

`--obsolete-only` replays the proofs only if the session contains obsolete proof attempts.

`--smoke-detector {none|top|deep}` tries to detect if the context is self-contradicting.

`--prover <prover>` restricts the replay to the selected provers only.

Smoke detector The smoke detector tries to detect if the context is self-contradicting and, thus, that anything can be proved in this context. The smoke detector can't be run on an outdated session and does not modify the session. It has three possible configurations:

`none` Do not run the smoke detector.

`top` The negation of each proved goal is sent with the same timeout to the prover that proved the original goal.

```
Goal G : forall x:int. q x -> (p1 x \/ p2 x)
```

becomes

```
Goal G : ~ (forall x:int. q x -> (p1 x \/ p2 x))
```

In other words, if the smoke detector is triggered, it means that the context of the goal `G` is self-contradicting.

`deep` This is the same technique as `top` but the negation is pushed under the universal quantification (without changing them) and under the implication. The previous example becomes

```
Goal G : forall x:int. q x /\ ~ (p1 x \/ p2 x)
```

In other words, the premises of goal `G` are pushed in the context, so that if the smoke detector is triggered, it means that the context of the goal `G` and its premises are self-contradicting. It should be clear that detecting smoke in that case does not necessarily means that there is a mistake: for example, this could occur in the WP of a program with an unfeasible path.

At the end of the replay, the name of the goals that triggered the smoke detector are printed:

```
goal 'G', prover 'Alt-Ergo 0.93.1': Smoke detected!!!
```

Moreover `Smoke detected` (exit code 1) is printed at the end if the smoke detector has been triggered, or `No smoke detected` (exit code 0) otherwise.

5.5 The `session` Command

The `session` command makes it possible to extract information from proof sessions on the command line, or even modify them to some extent. The invocation of this program is done under the form

```
why3 session <subcommand> [options] <session directories>
```

The available subcommands are as follows:

`info` prints informations and statistics about sessions.

`latex` outputs session contents in LaTeX format.

`html` outputs session contents in HTML format.

`update` update session contents.

The first three commands do not modify the sessions, whereas the last modify them.

5.5.1 Command `info`

The command `why3 session info` reports various informations about the session, depending on the following specific options.

`--provers` prints the provers that appear inside the session, one by line.

`--edited-files` prints all the files that appear in the session as edited proofs.

`--stats` prints various proofs statistics, as detailed below.

`--print0` separates the results of the options `provers` and `--edited-files` by the character number 0 instead of end of line `\n`. That allows you to safely use (even if the filename contains space or carriage return) the result with other commands. For example you can count the number of proof line in all the coq edited files in a session with:

```
why3 session info --edited-files vstte12_bfs --print0 | xargs -0 coqwc
```

or you can add all the edited files in your favorite repository with:

```
why3 session info --edited-files --print0 vstte12_bfs.mlw | \
    xargs -0 git add
```

Session Statistics The proof statistics given by option `--stats` are as follows:

- Number of goals: give both the total number of goals, and the number of those that are proved (possibly after a transformation).
- Goals not proved: list of goals of the session which are not proved by any prover, even after a transformation.
- Goals proved by only one prover: the goals for which there is only one successful proof. For each of these, the prover which was successful is printed. This also includes the sub-goals generated by transformations.
- Statistics per prover: for each of the prover used in the session, the number of proved goals is given. This also includes the sub-goals generated by transformations. The respective minimum, maximum and average time and on average running time is shown. Beware that these time data are computed on the goals *where the prover was successful*.

For example, here are the session statistics produced on the "hello proof" example of Section 1.

```
== Number of root goals ==
  total: 3  proved: 2

== Number of sub goals ==
  total: 2  proved: 1

== Goals not proved ==
  +-- file ../hello_proof.why
    +-- theory HelloProof
      +-- goal G2
        +-- transformation split_goal_right
          +-- goal G2.0

== Goals proved by only one prover ==
  +-- file ../hello_proof.why
    +-- theory HelloProof
      +-- goal G1: Alt-Ergo 0.99.1
      +-- goal G2
        +-- transformation split_goal_right
          +-- goal G2.1: Alt-Ergo 0.99.1
      +-- goal G3: Alt-Ergo 0.99.1

== Statistics per prover: number of proofs, time (minimum/maximum/average) in seconds ==
  Alt-Ergo 0.99.1     :   3   0.00   0.00   0.00
```

5.5.2 Command `latex`

Command `latex` produces a summary of the replay under the form of a tabular environment in LaTeX, one tabular for each theory, one per file.

The specific options are

`-style <n>` sets output style (1 or 2, default 1) Option `-style 2` produces an alternate version of LaTeX output, with a different layout of the tables.

`-o <dir>` indicates where to produce LaTeX files (default: the session directory).

`-longtable` uses the 'longtable' environment instead of 'tabular'.

`-e <elem>` produces a table for the given element, which is either a file, a theory or a root goal. The element must be specified using its path in dot notation, *e.g.* `file.theory.goal`. The file produced is named accordingly, *e.g.* `file.theory.goal.tex`. This option can be given several times to produce several tables in one run. When this option is given at least once, the default behavior that is to produce one table per theory is disabled.

Customizing LaTeX output The generated LaTeX files contain some macros that must be defined externally. Various definitions can be given to them to customize the output.

`\provername` macro with one parameter, a prover name

`\valid` macro with one parameter, used where the corresponding prover answers that the goal is valid. The parameter is the time in seconds.

`\noresult` macro without parameter, used where no result exists for the corresponding prover

`\timeout` macro without parameter, used where the corresponding prover reached the time limit

`\explanation` macro with one parameter, the goal name or its explanation

Figure 5.3 suggests some definitions for these macros, while Figures 5.4 and 5.5 show the tables obtained from the HelloProof example of Section 1, respectively with style 1 and 2.

5.5.3 Command `html`

This command produces a summary of the proof session in HTML syntax. There are two styles of output: 'table' and 'simpletree'. The default is 'table'.

The file generated is named `why3session.html` and is written in the session directory by default (see option `-o` to override this default).

The style 'table' outputs the contents of the session as a table, similar to the LaTeX output above. Figure 5.6 is the HTML table produced for the 'HelloProof' example, as typically shown in a Web browser. The gray cells filled with `--` just mean that the prover was not run on the corresponding goal. Green background means the result was "Valid", other cases are in orange background. The red background for a goal means that the goal was not proved.

```
\usepackage{xcolor}
\usepackage{colortbl}
\usepackage{rotating}

\newcommand{\provername}[1]{\cellcolor{yellow!25}
\begin{sideways}\textbf{#1}~~\end{sideways}}
\newcommand{\explanation}[1]{\cellcolor{yellow!13}lemma \texttt{#1}}
\newcommand{\transformation}[1]{\cellcolor{yellow!13}transformation \texttt{#1}}
\newcommand{\subgoal}[2]{\cellcolor{yellow!13}subgoal #2}
\newcommand{\valid}[1]{\cellcolor{green!13}#1}
\newcommand{\unknown}[1]{\cellcolor{red!20}#1}
\newcommand{\invalid}[1]{\cellcolor{red!50}#1}
\newcommand{\timeout}[1]{\cellcolor{red!20}(#1)}
\newcommand{\outofmemory}[1]{\cellcolor{red!20}(#1)}
\newcommand{\noresult}{\multicolumn{1}{>{\columncolor[gray]{0.8}}c|}{~}}
\newcommand{\failure}{\cellcolor{red!20}failure}
\newcommand{\highfailure}{\cellcolor{red!50}FAILURE}
```

Figure 5.3: Sample macros for the LaTeX command

Proof obligations		Alt-Ergo 0.99.1	Coq 8.7.1
lemma G1		0.00	
lemma G2		0.00	
	lemma G2.0	0.00	0.29
	lemma G2.1	0.00	
lemma G3		0.00	

Figure 5.4: LaTeX table produced for the HelloProof example (style 1)

The style 'simpletree' displays the contents of the session under the form of tree, similar to the tree view in the IDE. It uses only basic HTML tags such as `<ul>` and `<li>`.

Specific options for this command are as follows.

`--style <style>` sets the style to use, among `simpletree` and `table`; defaults to `table`.

`-o <dir>` sets the directory where to output the produced files ('`-`' for stdout). The default is to output in the same directory as the session itself.

`--context` adds context around the generated code in order to allow direct visualization (header, css, ...). It also adds in the output directory all the needed external files. It can't be set with stdout output.

`--add_pp <suffix> <cmd> <out_suffix>` sets a specific pretty-printer for files with the given suffix. Produced files use `<out_suffix>` as suffix. `<cmd>` must contain '`%i`'

Proof obligations	Alt-Ergo 0.99.1	Coq 8.7.1
lemma G1	0.00	
lemma G2	0.00	
transformation split_goal_right		
subgoal 1	0.00	0.29
subgoal 2	0.00	
lemma G3	0.00	

Figure 5.5: LaTeX table produced for the HelloProof example (style 2)

Why3 Proof Results for Project "hello_proof"

Theory "hello_proof.HelloProof": not fully verified

Obligations	Alt-Ergo 0.99.1	Coq 8.7.1
G1	0.00	---
G2	0.00	---
split_goal_right		
G2.0	0.00	0.29
G2.1	0.00	---
G3	0.00	---

Figure 5.6: HTML table produced for the HelloProof example

which will be replaced by the input file and '`%o`' which will be replaced by the output file.

`--coqdoc` uses the `coqdoc` command to display Coq proof scripts. This is equivalent to `--add_pp .v "coqdoc --no-index --html -o %o %i" .html`

5.5.4 Command `update`

The command `why3 session update` permits to modify the session contents, depending on the following specific options.

`-rename-file <src> <dst>` renames the file <*src*> to <*dst*> in the session. The file <*src*> itself is also renamed to <*dst*> in your filesystem.

5.6 The `doc` Command

This tool can produce HTML pages from Why3 source code. Why3 code for theories or modules is output in preformatted HTML code. Comments are interpreted in three different ways.

- Comments starting with at least three stars are completed ignored.
- Comments starting with two stars are interpreted as textual documentation. Special constructs are interpreted as described below. When the previous line is not empty, the comment is indented to the right, so as to be displayed as a description of that line.
- Comments starting with one star only are interpreted as code comments, and are typeset as the code

Additionally, all the Why3 identifiers are typeset with links so that one can navigate through the HTML documentation, going from some identifier use to its definition.

Options

`-o <dir>` defines the directory where to output the HTML files.

`--output <dir>` is the same as `-o`.

`--index` generates an index file `index.html`. This is the default behavior if more than one file is passed on the command line.

`--no-index` prevents the generation of an index file.

`--title <title>` sets title of the index page.

`--stdlib-url <url>` sets a URL for files found in load path, so that links to definitions can be added.

Typesetting textual comments Some constructs are interpreted:

- `{c text}` interprets character *c* as some typesetting command:

  1-6 a heading of level 1 to 6 respectively

  h raw HTML

- `‘code‘` is a code escape: the text *code* is typeset as Why3 code.

A CSS file `style.css` suitable for rendering is generated in the same directory as output files. This CSS style can be modified manually, since regenerating the HTML documentation will not overwrite an existing `style.css` file.

5.7 The `execute` Command

Why3 can symbolically execute programs written using the WhyML language (extension `.mlw`). See also Section 7.1.

5.8 The `extract` Command

Why3 can extract programs written using the WhyML language (extension `.mlw`) to OCaml. See also Section 7.2.

5.9 The realize Command

Why3 can produce skeleton files for proof assistants that, once filled, realize the given theories. See also Section 8.2.

5.10 The wc Command

Why3 can give some token statistics about WhyML source files.

Chapter 6

Language Reference

In this chapter, we describe the syntax and semantics of WhyML.

6.1 Lexical Conventions

Blank characters are space, horizontal tab, carriage return, and line feed. Blanks separate lexemes but are otherwise ignored. Comments are enclosed by `(*` and `*)` and can be nested. Note that `(*)` does not start a comment.

Strings are enclosed in double quotes (`"`). Double quotes can be escaped inside strings using the backslash character (`\`). The other special sequences are `\n` for line feed and `\t` for horizontal tab. In the following, strings are referred to with the non-terminal *string*.

The syntax for numerical constants is given by the following rules:

$$
\begin{array}{rcl}
\textit{digit} & ::= & \texttt{0} - \texttt{9} \\
\textit{hex-digit} & ::= & \texttt{0} - \texttt{9} \mid \texttt{a} - \texttt{f} \mid \texttt{A} - \texttt{F} \\
\textit{oct-digit} & ::= & \texttt{0} - \texttt{7} \\
\textit{bin-digit} & ::= & \texttt{0} \mid \texttt{1} \\
\textit{integer} & ::= & \textit{digit}\ (\textit{digit} \mid \texttt{\_})^* \\
 & \mid & (\texttt{0x} \mid \texttt{0X})\ \textit{hex-digit}\ (\textit{hex-digit} \mid \texttt{\_})^* \\
 & \mid & (\texttt{0o} \mid \texttt{0O})\ \textit{oct-digit}\ (\textit{oct-digit} \mid \texttt{\_})^* \\
 & \mid & (\texttt{0b} \mid \texttt{0B})\ \textit{bin-digit}\ (\textit{bin-digit} \mid \texttt{\_})^* \\
\textit{real} & ::= & \textit{digit}^+\ \textit{exponent} \\
 & \mid & \textit{digit}^+\ \texttt{.}\ \textit{digit}^*\ \textit{exponent}^? \\
 & \mid & \textit{digit}^*\ \texttt{.}\ \textit{digit}^+\ \textit{exponent}^? \\
 & \mid & (\texttt{0x} \mid \texttt{0X})\ \textit{hex-digit}^+\ \textit{h-exponent} \\
 & \mid & (\texttt{0x} \mid \texttt{0X})\ \textit{hex-digit}^+\ \texttt{.}\ \textit{hex-digit}^*\ \textit{h-exponent}^? \\
 & \mid & (\texttt{0x} \mid \texttt{0X})\ \textit{hex-digit}^*\ \texttt{.}\ \textit{hex-digit}^+\ \textit{h-exponent}^? \\
\textit{exponent} & ::= & (\texttt{e} \mid \texttt{E})\ (\texttt{-} \mid \texttt{+})^?\ \textit{digit}^+ \\
\textit{h-exponent} & ::= & (\texttt{p} \mid \texttt{P})\ (\texttt{-} \mid \texttt{+})^?\ \textit{digit}^+
\end{array}
$$

Integer and real constants have arbitrary precision. Integer constants can be given in base 10, 16, 8 or 2. Real constants can be given in base 10 or 16. Notice that the exponent in hexadecimal real constants is written in base 10.

Identifiers are composed of letters, digits, underscores, and primes. The syntax distinguishes identifiers that start with a lowercase letter or an underscore (*lident*), identifiers

that start with an uppercase letter (*uident*), and identifiers that start with a prime (*qident*, used exclusively for type variables):

$$\begin{aligned} \textit{alpha} &::= \texttt{a} - \texttt{z} \mid \texttt{A} - \texttt{Z} \\ \textit{suffix} &::= \textit{alpha} \mid \textit{digit} \mid \texttt{'} \mid \texttt{\_} \\ \textit{lident} &::= (\texttt{a} - \texttt{z})\ \textit{suffix}^* \mid \texttt{\_}\ \textit{suffix}^+ \\ \textit{uident} &::= (\texttt{A} - \texttt{Z})\ \textit{suffix}^* \\ \textit{qident} &::= \texttt{'}\ (\texttt{a} - \texttt{z})\ \textit{suffix}^* \end{aligned}$$

Identifiers that contain a prime followed by a letter, such as `int32'max`, are reserved for symbols introduced by Why3 and cannot be used for user-defined symbols.

In order to refer to symbols introduced in different namespaces (*scopes*), we can put a dot-separated "qualifier prefix" in front of an identifier (e.g. `Map.S.get`). This allows us to use the symbol `get` from the scope `Map.S` without importing it in the current namespace:

$$\begin{aligned} \textit{qualifier} &::= (\textit{uident}\ \texttt{.})^+ \\ \textit{lqualid} &::= \textit{qualifier}^?\ \textit{lident} \\ \textit{uqualid} &::= \textit{qualifier}^?\ \textit{uident} \end{aligned}$$

All parenthesised expressions in WhyML (types, patterns, logical terms, program expressions) admit a qualifier before the opening parenthesis, e.g. `Map.S.(get m i)`. This imports the indicated scope into the current namespace during the parsing of the expression under the qualifier. For the sake of convenience, the parentheses can be omitted when the expression itself is enclosed in parentheses, square brackets or curly braces.

Prefix and infix operators are built from characters organized in four precedence groups (*op-char-1* to *op-char-4*), with optional primes at the end:

$$\begin{aligned} \textit{op-char-1} &::= \texttt{=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{\~{}} \\ \textit{op-char-2} &::= \texttt{+} \mid \texttt{-} \\ \textit{op-char-3} &::= \texttt{*} \mid \texttt{/} \mid \texttt{\textbackslash} \mid \texttt{\%} \\ \textit{op-char-4} &::= \texttt{!} \mid \texttt{\$} \mid \texttt{\&} \mid \texttt{?} \mid \texttt{@} \mid \texttt{\^{}} \mid \texttt{.} \mid \texttt{:} \mid \texttt{|} \mid \texttt{\#} \\ \textit{op-char-1234} &::= \textit{op-char-1} \mid \textit{op-char-2} \mid \textit{op-char-3} \mid \textit{op-char-4} \\ \textit{op-char-234} &::= \textit{op-char-2} \mid \textit{op-char-3} \mid \textit{op-char-4} \\ \textit{op-char-34} &::= \textit{op-char-3} \mid \textit{op-char-4} \\ \textit{infix-op-1} &::= \textit{op-char-1234}^*\ \textit{op-char-1}\ \textit{op-char-1234}^*\ \texttt{'}^* \\ \textit{infix-op-2} &::= \textit{op-char-234}^*\ \textit{op-char-2}\ \textit{op-char-234}^*\ \texttt{'}^* \\ \textit{infix-op-3} &::= \textit{op-char-34}^*\ \textit{op-char-3}\ \textit{op-char-34}^*\ \texttt{'}^* \\ \textit{infix-op-4} &::= \textit{op-char-4}^+\ \texttt{'}^* \\ \textit{prefix-op} &::= \textit{op-char-1234}^+\ \texttt{'}^* \\ \textit{tight-op} &::= (\texttt{!} \mid \texttt{?})\ \textit{op-char-4}^*\ \texttt{'}^* \end{aligned}$$

Infix operators from a high-numbered group bind stronger than the infix operators from a low-numbered group. For example, infix operator `.*.` from group 3 would have a higher precedence than infix operator `->-` from group 1. Prefix operators always bind stronger than infix operators. The so-called "tight operators" are prefix operators that

have even higher precedence than the juxtaposition (application) operator, allowing us to write expressions like `inv !x` without parentheses.

Finally, any identifier, term, formula, or expression in a WhyML source can be tagged either with a string *attribute* or a location:

attribute	::=	`[@` ... `]`	attribute
	|	`[#` *string* *digit*$^+$ *digit*$^+$ *digit*$^+$ `]`	location

An attribute cannot contain newlines or closing square brackets; leading and trailing spaces are ignored. A location consists of a file name in double quotes, a line number, and starting and ending character positions.

6.2 Type expressions

WhyML features an ML-style type system with polymorphic types, variants (sum types), and records that can have mutable fields. The syntax for type expressions is the following:

type	::=	*lqualid* *type-arg*$^+$	polymorphic type symbol
	|	*type* `->` *type*	mapping type (right-associative)
	|	*type-arg*	
type-arg	::=	*lqualid*	monomorphic type symbol (sort)
	|	*qident*	type variable
	|	`()`	unit type
	|	`(` *type* (`,` *type*)$^+$ `)`	tuple type
	|	`{` *type* `}`	snapshot type
	|	*qualifier*$^?$ `(` *type* `)`	type in a scope

Built-in types are `int` (arbitrary precision integers), `real` (real numbers), `bool`, the arrow type (also called the *mapping type*), and the tuple types. The empty tuple type is also called the *unit type* and can be written as `unit`.

Note that the syntax for type expressions notably differs from the usual ML syntax. In particular, the type of polymorphic lists is written `list 'a`, and not `'a list`.

Snapshot types are specific to WhyML, they denote the types of ghost values produced by pure logical functions in WhyML programs. A snapshot of an immutable type is the type itself: thus, `{int}` is the same as `int` and `{list 'a}` is the same as `list 'a`. A snapshot of a mutable type, however, represents a snapshot value which cannot be modified anymore. Thus, a snapshot array `a` of type `{array int}` can be read from (`a[42]` is accepted) but not written into (`a[42] <- 0` is rejected). Generally speaking, a program function that expects an argument of a mutable type will accept an argument of the corresponding snapshot type as long as it is not modified by the function.

6.3 Logical expressions: terms and formulas

A significant part of a typical WhyML source file is occupied by non-executable logical content intended for specification and proof: function contracts, assertions, definitions of logical functions and predicates, axioms, lemmas, etc.

Logical expressions are called *terms*. Boolean terms are called *formulas*. Internally, Why3 distinguishes the proper formulas (produced by predicate symbols, propositional connectives and quantifiers) and the terms of type `bool` (produced by Boolean variables

term	::=	*integer*	integer constant
	|	*real*	real constant
	|	`true` | `false`	Boolean constant
	|	`()`	empty tuple
	|	*qualid*	qualified identifier
	|	*qualifier*$^?$ `(` *term* `)`	term in a scope
	|	*qualifier*$^?$ `begin` *term* `end`	*idem*
	|	*tight-op term*	tight operator
	|	`{` *term-field*$^+$ `}`	record
	|	`{` *term* `with` *term-field*$^+$ `}`	record update
	|	*term* `.` *lqualid*	record field access
	|	*term* `[` *term* `]` `'`*	collection access
	|	*term* `[` *term* `<-` *term* `]` `'`*	collection update
	|	*term* `[` *term* `..` *term* `]` `'`*	collection slice
	|	*term* `[` *term* `..` `]` `'`*	right-open slice
	|	*term* `[` `..` *term* `]` `'`*	left-open slice
	|	*term term*$^+$	application
	|	*prefix-op term*	prefix operator
	|	*term infix-op*-4 *term*	infix operator 4
	|	*term infix-op*-3 *term*	infix operator 3
	|	*term infix-op*-2 *term*	infix operator 2
	|	*term* `at` *uident*	past value
	|	`old` *term*	initial value
	|	*term infix-op*-1 *term*	infix operator 1
	|	...	continued in Fig. 6.2
term-field	::=	*lqualid* `=` *term* `;`	field = value
qualid	::=	*qualifier*$^?$ (*lident-ext* | *uident*)	qualified identifier
lident-ext	::=	*lident*	lowercase identifier
	|	`(` *ident-op* `)`	operator identifier
	|	`(` *ident-op* `)` (`_` | `'`) *alpha suffix**	associated identifier
ident-op	::=	*infix-op*-1	infix operator 1
	|	*infix-op*-2	infix operator 2
	|	*infix-op*-3	infix operator 3
	|	*infix-op*-4	infix operator 4
	|	*prefix-op* `_`	prefix operator
	|	*tight-op* `_`$^?$	tight operator
	|	`[ ]` `'`*	collection access
	|	`[ <- ]` `'`*	collection update
	|	`[ ]` `'`* `<-`	in-place update
	|	`[ .. ]` `'`*	collection slice
	|	`[ _ .. ]` `'`*	right-open slice
	|	`[ .. _ ]` `'`*	left-open slice

Figure 6.1: WhyML terms (part I).

term	::=	...	see Fig. 6.1
	|	`not` *term*	negation
	|	*term* `/\` *term*	conjunction
	|	*term* `&&` *term*	asymmetric conjunction
	|	*term* `\/` *term*	disjunction
	|	*term* `||` *term*	asymmetric disjunction
	|	*term* `by` *term*	proof indication
	|	*term* `so` *term*	consequence indication
	|	*term* `->` *term*	implication
	|	*term* `<->` *term*	equivalence
	|	*term* `:` *type*	type cast
	|	*attribute*$^+$ *term*	attributes
	|	*term* (`,` *term*)$^+$	tuple
	|	*quantifier quant-vars triggers*$^?$ `.` *term*	quantifier
	|	...	continued in Fig. 6.3
quantifier	::=	`forall` | `exists`	
quant-vars	::=	*quant-cast* (`,` *quant-cast*)*	
quant-cast	::=	*binder*$^+$ (`:` *type*)$^?$	
binder	::=	`_` | *bound-var*	
bound-var	::=	*lident attribute**	
triggers	::=	`[` *trigger* (`|` *trigger*)* `]`	
trigger	::=	*term* (`,` *term*)*	

Figure 6.2: WhyML terms (part II).

and logical functions that return `bool`). However, this distinction is not enforced on the syntactical level, and Why3 will perform the necessary conversions behind the scenes.

The syntax of WhyML terms is given in Figures 6.1-6.3. The constructions are listed in the order of decreasing precedence. For example, as was mentioned above, tight operators have the highest precedence of all operators, so that `-p.x` denotes the negation of the record field `p.x`, whereas `!p.x` denotes the field `x` of a record stored in the reference `p`.

An operator in parentheses acts as an identifier referring to that operator, for example, in a definition. To distinguish between prefix and infix operators, an underscore symbol is appended at the end: for example, `(-)` refers to the binary subtraction and `(-_)` to the unary negation. Tight operators cannot be used as infix operators, and thus do not require disambiguation.

In addition to prefix and infix operators, WhyML supports several mixfix bracket operators to manipulate various collection types: dictionaries, arrays, sequences, etc. Bracket operators do not have any predefined meaning and may be used to denote access and update operations for various user-defined collection types. We can introduce multiple bracket operations in the same scope by disambiguating them with primes after the closing bracket: for example, `a[i]` may denote array access and `s[i]'` sequence access. Notice that the in-place update operator `a[i] <- v` cannot be used inside logical terms: all effectful operations are restricted to program expressions. To represent the result of a collection update, we should use a pure logical update operator `a[i <- v]` instead.

WhyML supports "associated" names for operators, obtained by adding a suffix after the parenthesised operator name. For example, an axiom that represents the specification of the infix operator `(+)` may be called `(+)'spec` or `(+)_spec`. As with normal identifiers, names with a letter after a prime, such as `(+)'spec`, can only be introduced by Why3, and not by the user in a WhyML source.

The `at` and `old` operators are used inside postconditions and assertions to refer to the value of a mutable program variable at some past moment of execution (see the next section for details). These operators have higher precedence than the infix operators from group 1 (*infix-op-1*): `old i > j` is parsed as `(old i) > j` and not as `old (i > j)`.

Infix operators from groups 2-4 are left-associative. Infix operators from group 1 are non-associative and can be chained. For example, the term `0 <= i < j < length a` is parsed as the conjunction of three inequalities `0 <= i`, `i < j`, and `j < length a`.

As with normal identifiers, we can put a qualifier over a parenthesised operator, e.g. `Map.S.([]) m i`. Also, as noted above, a qualifier can be put over a parenthesised term, and the parentheses can be omitted if the term is a record or a record update.

The propositional connectives in WhyML formulas are listed in Figure 6.2. The non-standard connectives — asymmetric conjunction (`&&`), asymmetric disjunction (`||`), proof indication (`by`), and consequence indication (`so`) — are used to control the goal-splitting transformations of Why3 and provide integrated proofs for WhyML assertions, postconditions, lemmas, etc. The semantics of these connectives follows the rules below:

- A proof task for `A && B` is split into separate tasks for `A` and `A -> B`. If `A && B` occurs as a premise, it behaves as a normal conjunction.
- A case analysis over `A || B` is split into disjoint cases `A` and `not A /\ B`. If `A || B` occurs as a goal, it behaves as a normal disjunction.
- An occurrence of `A by B` generates a side condition `B -> A` (the proof justifies the affirmation). When `A by B` occurs as a premise, it is reduced to `A` (the proof is discarded). When `A by B` occurs as a goal, it is reduced to `B` (the proof is verified).
- An occurrence of `A so B` generates a side condition `A -> B` (the premise justifies the conclusion). When `A so B` occurs as a premise, it is reduced to the conjunction `A /\ B` (we use both the premise and the conclusion). When `A so B` occurs as a goal, it is reduced to `A` (the premise is verified).

For example, full splitting of the goal `(A by (exists x. B so C)) && D` produces four subgoals: `exists x. B` (the premise is verified), `forall x. B -> C` (the premise justifies the conclusion), `(exists x. B /\ C) -> A` (the proof justifies the affirmation), and finally, `A -> D` (the proof of `A` is discarded and `A` is used to prove `D`).

The behaviour of the splitting transformations is further controlled by attributes `[@stop_split]` and `[@case_split]`. Consult Section 9.5.5 for details.

Among the propositional connectives, `not` has the highest precedence, `&&` has the same precedence as `/\` (weaker than negation), `||` has the same precedence as `\/` (weaker than conjunction), `by`, `so`, `->`, and `<->` all have the same precedence (weaker than disjunction). All binary connectives except equivalence are right-associative. Equivalence is non-associative and is chained instead: `A <-> B <-> C` is transformed into a conjunction of `A <-> B` and `B <-> C`. To reduce ambiguity, WhyML forbids to place a non-parenthesised implication at the right-hand side of an equivalence: `A <-> B -> C` is rejected.

In Figure 6.3, we find the more advanced term constructions: conditionals, let-bindings, pattern matching, and local function definitions, either via the `let-in` construction or the `fun` keyword. The pure logical functions defined in this way are called *mappings*; they are first-class values of "arrow" type τ_1 `->` τ_2.

term	::=	...	see Fig. 6.1 and 6.2
	|	`if` *term* `then` *term* `else` *term*	conditional
	|	`match` *term* `with` *term-case*$^+$ `end`	pattern matching
	|	`let` *pattern* `=` *term* `in` *term*	let-binding
	|	`let` *symbol param*$^+$ `=` *term* `in` *term*	mapping definition
	|	`fun` *param*$^+$ `->` *term*	unnamed mapping
term-case	::=	`|` *pattern* `->` *term*	
pattern	::=	*binder*	variable or '`_`'
	|	`()`	empty tuple
	|	`{` (*lqualid* `=` *pattern* `;`)$^+$ `}`	record pattern
	|	*uqualid pattern**	constructor
	|	`ghost` *pattern*	ghost sub-pattern
	|	*pattern* `as ghost`$^?$ *bound-var*	named sub-pattern
	|	*pattern* `,` *pattern*	tuple pattern
	|	*pattern* `|` *pattern*	"or" pattern
	|	*qualifier*$^?$ `(` *pattern* `)`	pattern in a scope
symbol	::=	*lident-ext attribute**	user-defined symbol
param	::=	*type-arg*	unnamed typed
	|	*binder*	(un)named untyped
	|	`(` `ghost`$^?$ *type* `)`	unnamed typed
	|	`(` `ghost`$^?$ *binder* `)`	(un)named untyped
	|	`(` `ghost`$^?$ *binder*$^+$ `:` *type* `)`	multi-variable typed

Figure 6.3: WhyML terms (part III).

The patterns are similar to those of OCaml, though the `when` clauses and numerical constants are not supported. Unlike in OCaml, `as` binds stronger than the comma: in the pattern `(`p_1`,`p_2 `as x)`, variable `x` is bound to the value matched by pattern p_2. Also notice the closing `end` after the `match-with` term. A `let-in` construction with a non-trivial pattern is translated as a `match-with` term with a single branch.

Inside logical terms, pattern matching must be exhaustive: WhyML rejects a term like `let Some x = o in ...`, where `o` is a variable of an option type. In program expressions, non-exhaustive pattern matching is accepted and a proof obligation is generated to show that the values not covered cannot occur in execution.

The syntax of parameters in user-defined operations—first-class mappings, top-level logical functions and predicates, and program functions—is rather flexible in WhyML. Like in OCaml, the user can specify the name of a parameter without its type and let the type be inferred from the definition. Unlike in OCaml, the user can also specify the type of the parameter without giving its name. This is convenient when the symbol declaration does not provide the actual definition or specification of the symbol, and thus only the type signature is of relevance. For example, one can declare an abstract binary function that adds an element to a set simply by writing `function add 'a (set 'a) : set 'a`. A standalone non-qualified lowercase identifier without attributes is treated as a type name when the definition is not provided, and as a parameter name otherwise.

Ghost patterns, ghost variables after `as`, and ghost parameters in function definitions are only used in program code, and not allowed in logical terms.

6.4 Program expressions

The syntax of program expressions is given in Figures 6.4-6.5. As before, the constructions are listed in the order of decreasing precedence. The rules for tight, prefix, infix, and bracket operators are the same as for logical terms. In particular, the infix operators from group 1 can be chained. Notice that binary operators `&&` and `||` denote here the usual lazy conjunction and disjunction, respectively.

expr	::=	*integer*	integer constant
	|	*real*	real constant
	|	`true` | `false`	Boolean constant
	|	`()`	empty tuple
	|	*qualid*	identifier in a scope
	|	*qualifier*$^?$ `(` *expr* `)`	expression in a scope
	|	*qualifier*$^?$ `begin` *expr* `end`	*idem*
	|	*tight-op expr*	tight operator
	|	`{` (*lqualid* `=` *expr* `;`)$^+$ `}`	record
	|	`{` *expr* `with` (*lqualid* `=` *expr* `;`)$^+$ `}`	record update
	|	*expr* `.` *lqualid*	record field access
	|	*expr* `[` *expr* `]` `'`*	collection access
	|	*expr* `[` *expr* `<-` *expr* `]` `'`*	collection update
	|	*expr* `[` *expr* `..` *expr* `]` `'`*	collection slice
	|	*expr* `[` *expr* `..` `]` `'`*	right-open slice
	|	*expr* `[` `..` *expr* `]` `'`*	left-open slice
	|	*expr expr*$^+$	application
	|	*prefix-op expr*	prefix operator
	|	*expr infix-op-4 expr*	infix operator 4
	|	*expr infix-op-3 expr*	infix operator 3
	|	*expr infix-op-2 expr*	infix operator 2
	|	*expr infix-op-1 expr*	infix operator 1
	|	`not` *expr*	negation
	|	*expr* `&&` *expr*	lazy conjunction
	|	*expr* `||` *expr*	lazy disjunction
	|	*expr* `:` *type*	type cast
	|	*attribute*$^+$ *expr*	attributes
	|	`ghost` *expr*	ghost expression
	|	*expr* (`,` *expr*)$^+$	tuple
	|	*expr* `<-` *expr*	assignment
	|	...	continued in Fig. 6.5

Figure 6.4: WhyML program expressions (part I).

Keyword `ghost` marks the expression as ghost code added for verification purposes. Ghost code is removed from the final code intended for execution, and thus cannot affect the computation of the program results nor the content of the observable memory.

Assignment updates in place a mutable record field or an element of a collection. The former can be done simultaneously on a tuple of values: `x.f, y.g <- a, b`. The latter form, `a[i] <- v`, amounts to a call of the ternary bracket operator `([]<-)` and cannot be used in a multiple assignment.

expr	::=	...	see Fig. 6.4
	|	*expr* *spec*$^+$	added specification
	|	`if` *expr* `then` *expr* (`else` *expr*)$^?$	conditional
	|	`match` *expr* `with` (`|` *pattern* `->` *expr*)$^+$ `end`	pattern matching
	|	*qualifier*$^?$ `begin` *spec*$^+$ *expr* `end`	abstract block
	|	*expr* `;` *expr*	sequence
	|	`let` *pattern* `=` *expr* `in` *expr*	let-binding
	|	`let` *fun-defn* `in` *expr*	local function
	|	`let rec` *fun-defn* (`with` *fun-defn*)* `in` *expr*	recursive function
	|	`fun` *param*$^+$ *spec** `->` *spec** *expr*	unnamed function
	|	`any` *result* *spec**	arbitrary value
fun-defn	::=	*fun-head* *spec** `=` *spec** *expr*	function definition
fun-head	::=	`ghost`$^?$ *kind*$^?$ *symbol* *param*$^+$ (`:` *result*)$^?$	function header
kind	::=	`function` | `predicate` | `lemma`	function kind
result	::=	*ret-type*	
	|	`(` *ret-type* (`,` *ret-type*)* `)`	
	|	`(` *ret-name* (`,` *ret-name*)* `)`	
ret-type	::=	`ghost`$^?$ *type*	unnamed result
ret-name	::=	`ghost`$^?$ *binder* `:` *type*	named result
spec	::=	`requires { ` *term* ` }`	pre-condition
	|	`ensures { ` *term* ` }`	post-condition
	|	`returns { (|` *pattern* `->` *term*)$^+$ `}`	post-condition
	|	`raises { (|` *pattern* `->` *term*)$^+$ `}`	exceptional post-c.
	|	`raises { ` *uqualid* (`,` *uqualid*)* `}`	raised exceptions
	|	`reads { ` *lqualid* (`,` *lqualid*)* `}`	external reads
	|	`writes { ` *path* (`,` *path*)* `}`	memory writes
	|	`alias { ` *alias* (`,` *alias*)* `}`	memory aliases
	|	`variant { ` *variant* (`,` *variant*)* `}`	termination variant
	|	`diverges`	may not terminate
	|	(`reads` | `writes` | `alias`) `{ }`	empty effect
path	::=	*lqualid* (`.` *lqualid*)*	`v.field1.field2`
alias	::=	*path* `with` *path*	`arg1 with result`
variant	::=	*term* (`with` *lqualid*)$^?$	variant + WF-order

Figure 6.5: WhyML program expressions (part II).

6.5 The Why3 Language

6.5.1 Terms

The syntax for terms is given in Figure 6.1. The various constructs have the following priorities and associativities, from lowest to greatest priority:

construct	associativity
`if then else` / `let in`	–
label	–
cast	–
infix-op level 1	left
infix-op level 2	left
infix-op level 3	left
infix-op level 4	left
prefix-op	–
function application	left
brackets / ternary brackets	–
bang-op	–

Note the curryfied syntax for function application, though partial application is not allowed (rejected at typing).

6.5.2 Formulas

The syntax for formulas is given Figure 6.6. The various constructs have the following priorities and associativities, from lowest to greatest priority:

construct	associativity
`if then else` / `let in`	–
label	–
`->` / `<->`	right
`by` / `so`	right
`\/` / `||`	right
`/\` / `&&`	right
`not`	–
infix level 1	left
infix level 2	left
infix level 3	left
infix level 4	left
prefix	–

Note that infix symbols of level 1 include equality (`=`) and disequality (`<>`).

Notice that there are two symbols for the conjunction: `/\` and `&&`, and similarly for disjunction. They are logically equivalent, but may be treated slightly differently by some transformations. For instance, `split` transforms the goal `A /\ B` into subgoals `A` and `B`, whereas it transforms `A && B` into subgoals `A` and `A -> B`. Similarly, it transforms `not (A || B)` into subgoals `not A` and `not ((not A) /\ B)`. The `by`/`so` connectives are proof indications. They are logically equivalent to their first argument, but may affect the result of some transformations. For instance, the `split_goal` transformations interpret those connectives as introduction of logical cuts (see 9.5.5 for details).

6.5.3 Theories

The syntax for theories is given on Figure 6.7 and 6.8.

Algebraic types

TO BE COMPLETED

formula	::=	`true` | `false`	
	|	*formula* `->` *formula*	implication
	|	*formula* `<->` *formula*	equivalence
	|	*formula* `/\` *formula*	conjunction
	|	*formula* `&&` *formula*	asymmetric conj.
	|	*formula* `\/` *formula*	disjunction
	|	*formula* `||` *formula*	asymmetric disj.
	|	*formula* `by` *formula*	proof indication
	|	*formula* `so` *formula*	consequence indication
	|	`not` *formula*	negation
	|	*lqualid*	symbol
	|	*prefix-op term*	
	|	*term infix-op term*	
	|	*lqualid* $term^+$	predicate application
	|	`if` *formula* `then` *formula*	
		`else` *formula*	conditional
	|	`let` *pattern* `=` *term* `in` *formula*	local binding
	|	`match` *term* (`,` *term*)$^+$ `with`	
		(`|` *formula-case*)$^+$ `end`	pattern matching
	|	*quantifier binders* (`,` *binders*)*	
		$triggers^?$ `.` *formula*	quantifier
	|	*label formula*	label
	|	`(` *formula* `)`	parentheses
quantifier	::=	`forall` | `exists`	
binders	::=	*lident*$^+$ `:` *type*	
triggers	::=	`[` *trigger* (`|` *trigger*)* `]`	
trigger	::=	*tr-term* (`,` *tr-term*)*	
tr-term	::=	*term* | *formula*	
formula-case	::=	*pattern* `->` *formula*	

Figure 6.6: Syntax for formulas.

Record types

TO BE COMPLETED

Range types

A declaration of the form `type r = < range a b >` defines a type that projects into the integer range `[a,b]`. Note that in order to make such a declaration the theory `int.Int` must be imported.

Why3 let you cast an integer literal in a range type (e.g. `(42:r)`) and will check at typing that the literal is in range. Defining such a range type r automatically introduces the following:

```
function  r'int r : int
constant  r'maxInt : int
```

theory ::= `theory` *uident-nq label** *decl** `end`

decl ::= `type` *type-decl* (`with` *type-decl*)*
| `constant` *constant-decl*
| `function` *function-decl* (`with` *logic-decl*)*
| `predicate` *predicate-decl* (`with` *logic-decl*)*
| `inductive` *inductive-decl* (`with` *inductive-decl*)*
| `coinductive` *inductive-decl* (`with` *inductive-decl*)*
| `axiom` *ident-nq* `:` *formula*
| `lemma` *ident-nq* `:` *formula*
| `goal` *ident-nq* `:` *formula*
| `use` *imp-exp tqualid* (`as` *uident*)$^?$
| `clone` *imp-exp tqualid* (`as` *uident*)$^?$ *subst*$^?$
| `namespace` `import`$^?$ *uident-nq decl** `end`

logic-decl ::= *function-decl*
| *predicate-decl*

constant-decl ::= *lident-nq label** `:` *type*
| *lident-nq label** `:` *type* `=` *term*

function-decl ::= *lident-nq label** *type-param** `:` *type*
| *lident-nq label** *type-param** `:` *type* `=` *term*

predicate-decl ::= *lident-nq label** *type-param**
| *lident-nq label** *type-param** `=` *formula*

inductive-decl ::= *lident-nq label** *type-param** `=`
`|`$^?$ *ind-case* (`|` *ind-case*)*

ind-case ::= *ident-nq label** `:` *formula*

imp-exp ::= (`import` | `export`)$^?$

subst ::= `with` (`,` *subst-elt*)$^+$

subst-elt ::= `type` *lqualid* `=` *lqualid*
| `function` *lqualid* `=` *lqualid*
| `predicate` *lqualid* `=` *lqualid*
| `namespace` (*uqualid* | `.`) `=` (*uqualid* | `.`)
| `lemma` *qualid*
| `goal` *qualid*

tqualid ::= *uident* | *ident* (`.` *ident*)* `.` *uident*

type-decl ::= *lident-nq label** (`'` *lident-nq label**)* *type-defn*

Figure 6.7: Syntax for theories (part 1).

type-defn	::=		abstract type
	|	`=` *type*	alias type
	|	`=` `|`$^{?}$ *type-case* (`|` *type-case*)*	algebraic type
	|	`=` `{` *record-field* (`;` *record-field*)* `}`	record type
	|	`<` `range` *integer integer* `>`	range type
	|	`<` `float` *integer integer* `>`	float type
type-case	::=	*uident label** *type-param**	
record-field	::=	*lident label** `:` *type*	
type-param	::=	`'` *lident*	
	|	*lqualid*	
	|	`(` *lident*$^{+}$ `:` *type* `)`	
	|	`(` *type* (`,` *type*)* `)`	
	|	`()`	

Figure 6.8: Syntax for theories (part 2).

```
constant  r'minInt : int
```

The function `r'int` projects a term of type `r` to its integer value. The two constants represent the high bound and low bound of the range respectively.

Unless specified otherwise with the meta `"keep:literal"` on `r`, the transformation *eliminate_literal* introduces an axiom

```
axiom r'axiom : forall i:r. r'minInt <= r'int i <= r'maxInt
```

and replaces all casts of the form `(42:r)` with a constant and an axiom as in:

```
constant rliteral7 : r
axiom rliteral7_axiom : r'int rliteral7 = 42
```

This type is used in the standard library in the theories `bv.BV8`, `bv.BV16`, `bv.BV32`, `bv.BV64`.

Floating-point Types

A declaration of the form `type f = < float` *`eb sb`* `>` defines a type of floating-point numbers as specified by the IEEE-754 standard [8]. Here the literal *`eb`* represents the number of bits in the exponent and the literal *`sb`* the number of bits in the significand (including the hidden bit). Note that in order to make such a declaration the theory `real.Real` must be imported.

Why3 let you cast a real literal in a float type (e.g. `(0.5:f)`) and will check at typing that the literal is representable in the format. Note that Why3 do not implicitly round a real literal when casting to a float type, it refuses the cast if the literal is not representable.

Defining such a type `f` automatically introduces the following:

```
predicate f'isFinite f
function  f'real f : real
constant  f'eb : int
constant  f'sb : int
```

As specified by the IEEE standard, float formats includes infinite values and also a special NaN value (Not-a-Number) to represent results of undefined operations such as 0/0. The predicate `f'isFinite` indicates whether its argument is neither infinite nor NaN. The function `f'real` projects a finite term of type `f` to its real value, its result is not specified for non finite terms.

Unless specified otherwise with the meta `"keep:literal"` on `f`, the transformation *eliminate_literal* will introduce an axiom

```
axiom f'axiom :
  forall x:f. f'isFinite x -> -. max_real <=. f'real x <=. max_real
```

where `max_real` is the value of the biggest finite float in the specified format. The transformation also replaces all casts of the form `(0.5:f)` with a constant and an axiom as in:

```
constant fliteral42 : f
axiom fliteral42_axiom : f'real fliteral42 = 0.5 /\ f'isFinite fliteral42
```

This type is used in the standard library in the theories `ieee_float.Float32` and `ieee_float.Float64`.

6.5.4 Files

A Why3 input file is a (possibly empty) list of theories.

file	::=	*theory**

spec	::=	*requires* | *ensures* | *returns* | *raises*
	|	*reads* | *writes* | *variant*
requires	::=	`requires {` *formula* `}`
ensures	::=	`ensures {` *formula* `}`
returns	::=	`returns {` `|`$^?$ *formula-case* (`|` *formula-case*)* `}`
reads	::=	`reads {` *term* (`,` *term*)* `}`
writes	::=	`writes {` *term* (`,` *term*)* `}`
raises	::=	`raises {` `|`$^?$ *raises-case* (`|` *raises-case*)* `}`
	|	`raises {` *uqualid* (`,` *uqualid*)* `}`
raises-case	::=	*uqualid pattern*$^?$ `->` *formula*
variant	::=	`variant {` *one-variant* (`,` *one-variant*)$^+$ `}`
one-variant	::=	*term* (`with` *variant-rel*)$^?$
variant-rel	::=	*lqualid*
invariant	::=	`invariant {` *formula* `}`
assertion	::=	(`assert` | `assume` | `check`) `{` *formula* `}`
	|	`absurd`

Figure 6.9: Specification clauses in programs.

6.6 The WhyML Language

6.6.1 Specification

The syntax for specification clauses in programs is given in Figure 6.9. Within specifications, terms are extended with new constructs `old` and `at`:

term	::=	...
	|	`old` *term*
	|	`at` *term* `'` *uident*

Within a postcondition, `old` t refers to the value of term t in the prestate. Within the scope of a code mark L, the term `at` t `'`L refers to the value of term t at the program point corresponding to L.

6.6.2 Expressions

The syntax for program expressions is given in Figure 6.10 and Figure 6.11.

In applications, arguments are evaluated from right to left. This includes applications of infix operators, with the only exception of lazy operators `&&` and `||` that evaluate from left to right, lazily.

6.6.3 Modules

The syntax for modules is given in Figure 6.12. Any declaration which is accepted in a

expr	::=	*integer*	integer constant
	|	*real*	real constant
	|	*lqualid*	symbol
	|	*prefix-op expr*	
	|	*expr infix-op expr*	
	|	`expr [ expr ]`	brackets
	|	`expr [ expr ] <-` *expr*	brackets assignment
	|	`expr [` *expr infix-op*-1 *expr* `]`	ternary brackets
	|	*expr* $expr^{+}$	function application
	|	`fun` $binder^{+}$ $spec^{*}$ `->` $spec^{*}$ *expr*	lambda abstraction
	|	`let rec` *rec-defn* `in` *expr*	recursive functions
	|	`let` *fun-defn* `in` *expr*	local function
	|	`if` *expr* `then` *expr* (`else` *expr*)$^{?}$	conditional
	|	*expr* `;` *expr*	sequence
	|	`loop` $invariant^{*}$ $variant^{?}$ *expr* `end`	infinite loop
	|	`while` *expr*	while loop
		`do` $invariant^{*}$ $variant^{?}$ *expr* `done`	
	|	`for` *lident* `=` *expr to-downto expr*	for loop
		`do` $invariant^{*}$ *expr* `done`	
	|	*assertion*	assertion
	|	`raise` *uqualid*	exception raising
	|	`raise (` *uqualid expr* `)`	
	|	`try` *expr* `with` (`|` *handler*)$^{+}$ `end`	exception catching
	|	`any` *type* $spec^{*}$	
	|	`abstract` *expr* $spec^{*}$	blackbox
	|	`let` *pattern* `=` *expr* `in` *expr*	local binding
	|	`match` *expr* (`,` *expr*)* `with`	pattern matching
		`|`$^{?}$ *expr-case* (`|` *expr-case*)* `end`	
	|	`(` *expr* (`,` *expr*)$^{+}$ `)`	tuple
	|	`{` *expr-field*$^{+}$ `}`	record
	|	*expr* `.` *lqualid*	field access
	|	*expr* `.` *lqualid* `<-` *expr*	field assignment
	|	`{` *expr* `with` *expr-field*$^{+}$ `}`	field update
	|	*expr* `:` *type*	cast
	|	`ghost` *expr*	ghost expression
	|	*label expr*	label
	|	`'` *uident* `:` *expr*	code mark
	|	`(` *expr* `)`	parentheses
expr-case	::=	*pattern* `->` *expr*	
expr-field	::=	*lqualid* `=` *expr* `;`	
handler	::=	*uqualid* $pattern^{?}$ `->` *expr*	
to-downto	::=	`to` | `downto`	

Figure 6.10: Syntax for program expressions (part 1).

expr	::=	...	see Fig. 6.4
	|	*expr* *spec*$^+$	added specification
	|	`if` *expr* `then` *expr* (`else` *expr*)$^?$	conditional
	|	`match` *expr* `with` (`|` *pattern* `->` *expr*)$^+$ `end`	pattern matching
	|	*qualifier*$^?$ `begin` *spec*$^+$ *expr* `end`	abstract block
	|	*expr* `;` *expr*	sequence
	|	`let` *pattern* `=` *expr* `in` *expr*	let-binding
	|	`let` *fun-defn* `in` *expr*	local function
	|	`let rec` *fun-defn* (`with` *fun-defn*)* `in` *expr*	recursive function
	|	`fun` *param*$^+$ *spec** `->` *spec** *expr*	unnamed function
	|	`any` *result* *spec**	arbitrary value
fun-defn	::=	*fun-head* *spec** `=` *spec** *expr*	function definition
fun-head	::=	`ghost`$^?$ *kind*$^?$ *symbol* *param*$^+$ (`:` *result*)$^?$	function header
kind	::=	`function` | `predicate` | `lemma`	function kind
result	::=	*ret-type*	
	|	`(` *ret-type* (`,` *ret-type*)* `)`	
	|	`(` *ret-name* (`,` *ret-name*)* `)`	
ret-type	::=	`ghost`$^?$ *type*	unnamed result
ret-name	::=	`ghost`$^?$ *binder* `:` *type*	named result
spec	::=	`requires { ` *term* ` }`	pre-condition
	|	`ensures { ` *term* ` }`	post-condition
	|	`returns { (|` *pattern* `->` *term*)$^+$ `}`	post-condition
	|	`raises { (|` *pattern* `->` *term*)$^+$ `}`	exceptional post-c.
	|	`raises { ` *uqualid* (`,` *uqualid*)* `}`	raised exceptions
	|	`reads { ` *lqualid* (`,` *lqualid*)* `}`	external reads
	|	`writes { ` *path* (`,` *path*)* `}`	memory writes
	|	`alias { ` *alias* (`,` *alias*)* `}`	memory aliases
	|	`variant { ` *variant* (`,` *variant*)* `}`	termination variant
	|	`diverges`	may not terminate
	|	(`reads` | `writes` | `alias`) `{ }`	empty effect
path	::=	*lqualid* (`.` *lqualid*)*	`v.field1.field2`
alias	::=	*path* `with` *path*	`arg1 with result`
variant	::=	*term* (`with` *lqualid*)$^?$	variant + WF-order

Figure 6.11: Syntax for program expressions (part 2).

module	::=	`module` *uident-nq label** *mdecl** `end`	
mdecl	::=	*decl*	theory declaration
	|	`type` *mtype-decl* (`with` *mtype-decl*)*	mutable types
	|	`type` *lident-nq* (`'` *lident-nq*)* *invariant*$^+$	added invariant
	|	`let ghost`$^?$ *lident-nq label** *pgm-defn*	
	|	`let rec` *rec-defn*	
	|	`val ghost`$^?$ *lident-nq label** *pgm-decl*	
	|	`exception` *lident-nq label** *type*$^?$	
	|	`namespace import`$^?$ *uident-nq mdecl** `end`	
mtype-decl	::=	*lident-nq label** (`'` *lident-nq label**)* *mtype-defn*	
mtype-defn	::=		abstract type
	|	`=` *type*	alias type
	|	`=` `|`$^?$ *type-case* (`|` *type-case*)* *invariant**	algebraic type
	|	`= {` *mrecord-field* (`;` *mrecord-field*)* `}` *invariant**	record type
mrecord-field	::=	`ghost`$^?$ `mutable`$^?$ *lident-nq label** `:` *type*	
pgm-defn	::=	*fun-body*	
	|	`= fun` *binder*$^+$ *spec** `->` *spec** *expr*	
pgm-decl	::=	`:` *type*	global variable
	|	*param* (*spec** *param*)$^+$ `:` *type spec**	abstract function

Figure 6.12: Syntax for modules.

theory is also accepted in a module. Additionally, modules can introduce record types with mutable fields and declarations which are specific to programs (global variables, functions, exceptions).

6.6.4 Files

A WhyML input file is a (possibly empty) list of theories and modules.

file	::=	(*theory* | *module*)*

A theory defined in a WhyML file can only be used within that file. If a theory is supposed to be reused from other files, be they Why3 or WhyML files, it should be defined in a Why3 file.

6.7 The Why3 Standard Library

The Why3 standard library provides general-purpose modules, to be used in logic and/or programs. It can be browsed on-line at http://why3.lri.fr/stdlib/. Each file contains one or several modules. To `use` or `clone` a module `M` from file `file`, use the syntax `file.M`, since `file` is available in Why3's default load path. For instance, the module of integers and the module of references are imported as follows:

```
use import int.Int
use import ref.Ref
```

A sub-directory `mach/` provides various modules to model machine arithmetic. For instance, the module of 63-bit integers and the module of arrays indexed by 63-bit integers are imported as follows:

```
use import mach.int.Int63
use import mach.array.Array63
```

In particular, the types and operations from these modules are mapped to native OCaml's types and operations when Why3 code is extracted to OCaml (see Section 7.2).

Chapter 7

Executing WhyML Programs

This chapter shows how WhyML code can be executed, either by being interpreted or compiled to some existing programming language.

Let us consider the program in Figure 2.1 on page 22 that computes the maximum and the sum of an array of integers. Let us assume it is contained in a file `maxsum.mlw`.

7.1 Interpreting WhyML Code

To test function `max_sum`, we can introduce a WhyML test function in module `MaxAndSum`

```
let test () =
  let n = 10 in
  let a = make n 0 in
  a[0] <- 9; a[1] <- 5; a[2] <- 0; a[3] <- 2;  a[4] <- 7;
  a[5] <- 3; a[6] <- 2; a[7] <- 1; a[8] <- 10; a[9] <- 6;
  max_sum a n
```

and then we use the `execute` command to interpret this function, as follows:

```
> why3 execute maxsum.mlw MaxAndSum.test
Execution of MaxAndSum.test ():
     type: (int, int)
   result: (45, 10)
  globals:
```

We get the expected output, namely the pair `(45, 10)`.

7.2 Compiling WhyML to OCaml

An alternative to interpretation is to compile WhyML to OCaml. We do so using the `extract` command, as follows:

```
> why3 extract -D ocaml64 maxsum.mlw -o max_sum.ml
```

The `extract` command requires the name of a driver, which indicates how theories/modules from the Why3 standard library are translated to OCaml. Here we assume a 64-bit architecture and thus we pass `ocaml64`. We also specify an output file using option `-o`, namely `max_sum.ml`. After this command, the file `max_sum.ml` contains an OCaml code for function `max_sum`. To compile it, we create a file `main.ml` containing a call to `max_sum`, *e.g.*,

```
let a = Array.map Z.of_int [| 9; 5; 0; 2; 7; 3; 2; 1; 10; 6 |]
let s, m = Max_sum.max_sum a (Z.of_int 10)
let () = Format.printf "sum=%s, max=%s@." (Z.to_string s) (Z.to_string m)
```

It is convenient to use `ocamlbuild` to compile and link both files `max_sum.ml` and `main.ml`:

```
> ocamlbuild -pkg zarith main.native
```

Since Why3's type `int` is translated to OCaml arbitrary precision integers using the `ZArith` library, we have to pass option `-pkg zarith` to `ocamlbuild`. In order to get extracted code that uses OCaml's native integers instead, one has to use Why3's types for 63-bit integers from libraries `mach.int.Int63` and `mach.array.Array63`.

Extraction Starting Point. The `extract` command accepts three different targets for extraction: a WhyML file, a module, or a symbol (function, type, exception). To extract all the symbols from every module of a file named `f.mlw`, one should write

```
> why3 extract -D <driver> f.mlw
```

To extract only the symbols from module `M` of file `f.mlw`, one should write

```
> why3 extract -D <driver> -L <dir> f.M
```

To extract only the symbol `s` (a function, a type, or an exception) from module `M` of file `f.mlw`, one should write

```
> why3 extract -D <driver> -L <dir> f.M.s
```

Note the use of `-L <dir>`, for both extraction of a module and a symbol, in order to state the location of file `f.mlw`.

Options. The following options can be added to the extraction command line:

`--flat` performs a flat extraction, *i.e.*, everything is extracted into a single file. This is the default behavior. The `-o` option should be given the name of a file or, if omitted, the result of extraction is printed to the standard output.

`--modular` each module is extracted in its own, separated file. The `-o` option cannot be omitted, and it should be given the name of an existing directory. This directory will be populated with the resulting OCaml files.

`--recursive` recursively extracts all the dependencies of the chosen entry point. This option is valid for both `modular` and `flat` options.

Examples. We illustrate different ways of using the `extract` command through some examples. Consider the program in Figure 2.6 on page 34. If we are only interested in extracting function `enqueue`, we can proceed as follows:

```
> why3 extract -D ocaml64 -L . aqueue.AmortizedQueue.enqueue -o aqueue.ml
```

Here we assume that file `aqueue.mlw` contains this program, and that we invoke `extract` from the directory where this file is stored. File `aqueue.ml` now contains the following OCaml code:

```
let enqueue (x: 'a) (q: 'a queue) : 'a queue =
  create (q.front) (q.lenf) (x :: (q.rear))
    (Z.add (q.lenr) (Z.of_string "1"))
```

Choosing a function symbol as the entry point of extraction allows us to focus only on specific parts of the program. However, the generated code cannot be type-checked by the OCaml compiler, as it depends on function `create` and on type `'a queue`, whose definitions are not given. In order to obtain a *complete* OCaml implementation, we can perform a recursive extraction:

```
> why3 extract --recursive -D ocaml64 -L . \
    aqueue.AmortizedQueue.enqueue -o aqueue.ml
```

This updates the contents of file `aqueue.ml` as follows:

```
type 'a queue = {
  front: 'a list;
  lenf: Z.t;
  rear: 'a list;
  lenr: Z.t;
  }

let create (f: 'a list) (lf: Z.t) (r: 'a list) (lr: Z.t) : 'a queue =
  if Z.geq lf lr
  then
    { front = f; lenf = lf; rear = r; lenr = lr }
  else
    let f1 = List.append f (List.rev r) in
    { front = f1; lenf = Z.add lf lr; rear = []; lenr = (Z.of_string "0") }

let enqueue (x: 'a) (q: 'a queue) : 'a queue =
  create (q.front) (q.lenf) (x :: (q.rear))
    (Z.add (q.lenr) (Z.of_string "1"))
```

This new version of the code is now accepted by the OCaml compiler (provided the `ZArith` library is available, as above).

Custom Extraction Drivers. Several OCaml drivers can be specified on the command line, using option `-D` several times. In particular, one can provide a custom driver to map some symbols of a Why3 development to existing OCaml code. Suppose for instance we have a file `file.mlw` containing a proof parameterized with some type `elt` and some binary function `f`:

```
module M
  type elt
  val f (x y: elt) : elt
  let double (x: elt) : elt = f x x
  ...
```

When it comes to extract this module to OCaml, we may want to instantiate type `elt` with OCaml's type `int` and function `f` with OCaml's addition. For this purpose, we provide the following in a file `mydriver.drv`:

```
module file.M
  syntax type elt "int"
  syntax val  f   "%1 + %2"
end
```

OCaml fragments to be substituted for Why3 symbols are given as arbitrary strings, where `%1`, `%2`, etc., will be replaced with actual arguments. Here is the extraction command line and its output:

```
> why3 extract -D ocaml64 -D mydriver.drv -L . file.M
let double (x: int) : int = x + x
...
```

When using such custom drivers, it is not possible to pass Why3 file names on the command line; one has to specify module names to be extracted, as done above.

Chapter 8

Interactive Proof Assistants

8.1 Using an Interactive Proof Assistant to Discharge Goals

Instead of calling an automated theorem prover to discharge a goal, Why3 offers the possibility to call an interactive theorem prover instead. In that case, the interaction is decomposed into two distinct phases:

- Edition of a proof script for the goal, typically inside a proof editor provided by the external interactive theorem prover;
- Replay of an existing proof script.

An example of such an interaction is given in the tutorial section 1.2.

Some proof assistants offer more than one possible editor, *e.g.* a choice between the use of a dedicated editor and the use of the Emacs editor and the ProofGeneral mode. Selection of the preferred mode can be made in `why3ide` preferences, under the "Editors" tab.

8.2 Theory Realizations

Given a Why3 theory, one can use a proof assistant to make a *realization* of this theory, that is to provide definitions for some of its uninterpreted symbols and proofs for some of its axioms. This way, one can show the consistency of an axiomatized theory and/or make a connection to an existing library (of the proof assistant) to ease some proofs.

8.2.1 Generating a realization

Generating the skeleton for a theory is done by passing to the `realize` command a driver suitable for realizations, the names of the theories to realize, and a target directory.

```
why3 realize -D path/to/drivers/prover-realize.drv
             -T env_path.theory_name -o path/to/target/dir/
```

The theory is looked into the files from the environment, *e.g.* the standard library. If the theory is stored in a different location, option `-L` should be used.

The name of the generated file is inferred from the theory name. If the target directory already contains a file with the same name, Why3 extracts all the parts that it assumes to be user-edited and merges them in the generated file.

Note that Why3 does not track dependencies between realizations and theories, so a realization will become outdated if the corresponding theory is modified. It is up to the user to handle such dependencies, for instance using a `Makefile`.

8.2.2 Using realizations inside proofs

If a theory has been realized, the Why3 printer for the corresponding prover will no longer output declarations for that theory but instead simply put a directive to load the realization. In order to tell the printer that a given theory is realized, one has to add a meta declaration in the corresponding theory section of the driver.

```
theory env_path.theory_name
  meta "realized_theory" "env_path.theory_name", "optional_naming"
end
```

The first parameter is the theory name for Why3. The second parameter, if not empty, provides a name to be used inside generated scripts to point to the realization, in case the default name is not suitable for the interactive prover.

8.2.3 Shipping libraries of realizations

While modifying an existing driver file might be sufficient for local use, it does not scale well when the realizations are to be shipped to other users. Instead, one should create two additional files: a configuration file that indicates how to modify paths, provers, and editors, and a driver file that contains only the needed `meta "realized_theory"` declarations. The configuration file should be as follows.

```
[main]
loadpath="path/to/theories"

[prover_modifiers]
name="Coq"
option="-R path/to/vo/files Logical_directory"
driver="path/to/file/with/meta.drv"

[editor_modifiers coqide]
option="-R path/to/vo/files Logical_directory"

[editor_modifiers proofgeneral-coq]
option="--eval \"(setq coq-load-path (cons '(\\\"path/to/vo/files\\\" \
  \\\"Logical_directory\\\") coq-load-path))\""
```

This configuration file can be passed to Why3 thanks to the `--extra-config` option.

8.3 Coq

This section describes the content of the Coq files generated by Why3 for both proof obligations and theory realizations. When reading a Coq script, Why3 is guided by the presence of empty lines to split the script, so the user should refrain from removing empty lines around generated blocks or adding empty lines inside them.

1. The header of the file contains all the library inclusions required by the driver file. Any user-made changes to this block will be lost when the file is regenerated by Why3. This part starts with `(* This file is generated by ... *)`.

2. Abstract logic symbols are assumed with the vernacular directive `Parameter`. Axioms are assumed with the `Axiom` directive. When regenerating a script, Why3 assumes that all such symbols have been generated by a previous run. As a consequence, the user should not introduce new symbols with these two directives, as they would be lost.

3. Definitions of functions and inductive types in theories are printed in a block that starts with `(* Why3 assumption *)`. This comment should not be removed; otherwise Why3 will assume that the definition is a user-defined block.

4. Proof obligations and symbols to be realized are introduced by `(* Why3 goal *)`. The user is supposed to fill the script after the statement. Why3 assumes that the user-made part extends up to `Qed`, `Admitted`, `Save`, or `Defined`, whichever comes first. In the case of definitions, the original statement can be replaced by a `Notation` directive, in order to ease the usage of predefined symbols. Why3 also recognizes `Variable` and `Hypothesis` and preserves them; they should be used in conjunction with Coq's `Section` mechanism to realize theories that still need some abstract symbols and axioms.

Why3 preserves any block from the script that does not fall into one of the previous categories. Such blocks can be used to import other libraries than the ones from the prelude. They can also be used to state and prove auxiliary lemmas. Why3 tries to preserve the position of these user-defined blocks relatively to the generated ones.

Currently, the parser for Coq scripts is rather naive and does not know much about comments. For instance, Why3 can easily be confused by some terminating directive like `Qed` that would be present in a comment.

8.4 Isabelle/HOL

When using Isabelle from Why3, files generated from Why3 theories and goals are stored in a dedicated XML format. Those files should not be edited. Instead, the proofs must be completed in a file with the same name and extension `.thy`. This is the file that is opened when using "Edit" action in `why3 ide`.

8.4.1 Installation

You need version Isabelle2017 or Isabelle2018. Former versions are not supported. We assume below that your version is 2018, please replace 2018 by 2017 otherwise.

Isabelle must be installed before compiling Why3. After compilation and installation of Why3, you must manually add the path

```
<Why3 lib dir>/isabelle
```

into either the user file

```
.isabelle/Isabelle2018/etc/components
```

or the system-wide file

```
<Isabelle install dir>/etc/components
```

8.4.2 Usage

The most convenient way to call Isabelle for discharging a Why3 goal is to start the Isabelle/jedit interface in server mode. In this mode, one must start the server once, before launching `why3 ide`, using

```
isabelle why3_jedit
```

Then, inside a `why3 ide` session, any use of "Edit" will transfer the file to the already opened instance of jEdit. When the proof is completed, the user must send back the edited proof to `why3 ide` by closing the opened buffer, typically by hitting `Ctrl-w`.

8.4.3 Using Isabelle 2018 server

Starting from Isabelle version 2018, Why3 is able to exploit the server features of Isabelle to speed up the processing of proofs in batch mode, e.g. when replaying them from within Why3 IDE. Currently, when replaying proofs using the `isabelle why3` tool, an Isabelle process including a rather heavyweight Java/Scala and PolyML runtime environment has to be started, and a suitable heap image has to be loaded for each proof obligation, which can take several seconds. To avoid this overhead, an Isabelle server and a suitable session can be started once, and then `isabelle why3` can just connect to it and request the server to process theories. In order to allow a tool such as Why3 IDE to use the Isabelle server, it has to be started via the wrapper tool `isabelle use_server`. For example, to process the proofs in examples/logic/genealogy using Why3 IDE and the Isabelle server, do the following:

1. Start an Isabelle server using

   ```
   isabelle server &
   ```

2. Start Why3 IDE using

   ```
   isabelle use_server why3 ide genealogy
   ```

8.4.4 Realizations

Realizations must be designed in some `.thy` as follows. The realization file corresponding to some Why3 file `f.why` should have the following form.

```
theory Why3_f
imports Why3_Setup
begin

section {* realization of theory T *}

why3_open "f/T.xml"

why3_vc <some lemma>
<proof>

why3_vc <some other lemma> by proof
```

```
[...]

why3_end
```

See directory `lib/isabelle` for examples.

8.5 PVS

8.5.1 Installation

You need version 6.0.

8.5.2 Usage

When a PVS file is regenerated, the old version is split into chunks, according to blank lines. Chunks corresponding to Why3 declarations are identified with a comment starting with `% Why3`, *e.g.*

```
% Why3 f
f(x: int) : int
```

Other chunks are considered to be user PVS declarations. Thus a comment such as `% Why3 f` must not be removed; otherwise, there will be two declarations for `f` in the next version of the file (one being regenerated and another one considered to be a user-edited chunk).

8.5.3 Realization

The user is allowed to perform the following actions on a PVS realization:

- give a definition to an uninterpreted symbol (type, function, or predicate symbol), by adding an equal sign (`=`) and a right-hand side to the definition. When the declaration is regenerated, the left-hand side is updated and the right-hand side is reprinted as is. In particular, the names of a function or predicate arguments should not be modified. In addition, the `MACRO` keyword may be inserted and it will be kept in further generations.
- turn an axiom into a lemma, that is to replace the PVS keyword `AXIOM` with either `LEMMA` or `THEOREM`.
- insert anything between generated declarations, such as a lemma, an extra definition for the purpose of a proof, an extra `IMPORTING` command, etc. Do not forget to surround these extra declarations with blank lines.

Why3 makes some effort to merge new declarations with old ones and with user chunks. If it happens that some chunks could not be merged, they are appended at the end of the file, in comments.

Chapter 9

Technical Informations

9.1 Structure of Session Files

The proof session state is stored in an XML file named `<dir>/why3session.xml`, where `<dir>` is the directory of the project. The XML file follows the DTD given in `share/why3session.dtd` and reproduced below.

```
<!ELEMENT why3session (prover*, file*)>
<!ATTLIST why3session shape_version CDATA #IMPLIED>

<!ELEMENT prover EMPTY>
<!ATTLIST prover id CDATA #REQUIRED>
<!ATTLIST prover name CDATA #REQUIRED>
<!ATTLIST prover version CDATA #REQUIRED>
<!ATTLIST prover alternative CDATA #IMPLIED>
<!ATTLIST prover timelimit CDATA #IMPLIED>
<!ATTLIST prover memlimit CDATA #IMPLIED>
<!ATTLIST prover steplimit CDATA #IMPLIED>

<!ELEMENT file (path*, theory*)>
<!ATTLIST file name CDATA #IMPLIED>
<!ATTLIST file verified CDATA #IMPLIED>
<!ATTLIST file proved CDATA #IMPLIED>

<!ELEMENT path EMPTY>
<!ATTLIST path name CDATA #REQUIRED>

<!ELEMENT theory (label*,goal*)>
<!ATTLIST theory name CDATA #REQUIRED>
<!ATTLIST theory verified CDATA #IMPLIED>
<!ATTLIST theory proved CDATA #IMPLIED>

<!ELEMENT goal (label*, proof*, transf*)>
<!ATTLIST goal name CDATA #REQUIRED>
<!ATTLIST goal expl CDATA #IMPLIED>
<!ATTLIST goal sum CDATA #IMPLIED>
<!ATTLIST goal shape CDATA #IMPLIED>
<!ATTLIST goal proved CDATA #IMPLIED>
```

```
<!ELEMENT proof (result|undone|internalfailure|unedited)>
<!ATTLIST proof prover CDATA #REQUIRED>
<!ATTLIST proof timelimit CDATA #IMPLIED>
<!ATTLIST proof memlimit CDATA #IMPLIED>
<!ATTLIST proof steplimit CDATA #IMPLIED>
<!ATTLIST proof edited CDATA #IMPLIED>
<!ATTLIST proof obsolete CDATA #IMPLIED>

<!ELEMENT result EMPTY>
<!ATTLIST result status (valid|invalid|unknown|timeout|outofmemory|steplimitexceeded|fail
<!ATTLIST result time CDATA #IMPLIED>
<!ATTLIST result steps CDATA #IMPLIED>

<!ELEMENT undone EMPTY>
<!ELEMENT unedited EMPTY>

<!ELEMENT internalfailure EMPTY>
<!ATTLIST internalfailure reason CDATA #REQUIRED>

<!ELEMENT transf (goal*)>
<!ATTLIST transf name CDATA #REQUIRED>
<!ATTLIST transf proved CDATA #IMPLIED>
<!ATTLIST transf arg1 CDATA #IMPLIED>
<!ATTLIST transf arg2 CDATA #IMPLIED>
<!ATTLIST transf arg3 CDATA #IMPLIED>
<!ATTLIST transf arg4 CDATA #IMPLIED>

<!ELEMENT label EMPTY>
<!ATTLIST label name CDATA #REQUIRED>
```

9.2 Prover Detection

The data configuration for the automatic detection of installed provers is stored in the file `provers-detection-data.conf` typically located in directory `/usr/local/share/why3` after installation. The content of this file is reproduced below.

```
[ATP alt-ergo]
name = "Alt-Ergo"
exec = "alt-ergo"
exec = "alt-ergo-2.3.0"
exec = "alt-ergo-2.2.0"
exec = "alt-ergo-2.1.0"
exec = "alt-ergo-2.0.0"
version_switch = "-version"
version_regexp = "^\\([0-9.]+\\)$"
version_ok = "2.3.0"
version_ok = "2.2.0"
version_ok = "2.1.0"
version_ok = "2.0.0"
version_bad = "1.30"
version_bad = "1.01"
version_bad = "0.99.1"
```

```
version_bad = "0.95.2"
command = "%e -timelimit %t %f"
command_steps = "%e -steps-bound %S %f"
driver = "alt_ergo"
editor = "altgr-ergo"
use_at_auto_level = 1

# CVC4 version >= 1.6, with counterexamples
[ATP cvc4-ce]
name = "CVC4"
alternative = "counterexamples"
exec = "cvc4"
exec = "cvc4-1.6"
exec = "cvc4-1.7"
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
version_ok = "1.6"
version_ok = "1.7"
driver = "cvc4_16_counterexample"
# --random-seed=42 is not needed as soon as --random-freq=0.0 by default
# to try: --inst-when=full-last-call
command = "%e --tlimit-per=%t000 --lang=smt2 %f"
command_steps = "%e --stats --rlimit=%S --lang=smt2 %f"

# CVC4 version >= 1.6
[ATP cvc4]
name = "CVC4"
exec = "cvc4"
exec = "cvc4-1.6"
exec = "cvc4-1.7"
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
version_ok = "1.6"
version_ok = "1.7"
driver = "cvc4_16"
# --random-seed=42 is not needed as soon as --random-freq=0.0 by default
# to try: --inst-when=full-last-call
command = "%e --tlimit=%t000 --lang=smt2 %f"
command_steps = "%e --stats --rlimit=%S --lang=smt2 %f"
use_at_auto_level = 1

# CVC4 version = 1.5, with counterexamples
[ATP cvc4-ce]
name = "CVC4"
alternative = "counterexamples"
exec = "cvc4"
exec = "cvc4-1.5"
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
version_ok = "1.5"
driver = "cvc4_15_counterexample"
# --random-seed=42 is not needed as soon as --random-freq=0.0 by default
# to try: --inst-when=full-last-call
command = "%e --tlimit-per=%t000 --lang=smt2 %f"
command_steps = "%e --stats --rlimit=%S --lang=smt2 %f"

# CVC4 version 1.5
[ATP cvc4]
name = "CVC4"
exec = "cvc4"
exec = "cvc4-1.5"
```

```
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
version_ok = "1.5"
driver = "cvc4_15"
# --random-seed=42 is not needed as soon as --random-freq=0.0 by default
# to try: --inst-when=full-last-call
command = "%e --tlimit=%t000 --lang=smt2 %f"
command_steps = "%e --stats --rlimit=%S --lang=smt2 %f"
use_at_auto_level = 1

# CVC4 version 1.4, using SMTLIB fixed-size bitvectors
[ATP cvc4]
name = "CVC4"
exec = "cvc4"
exec = "cvc4-1.4"
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
version_old = "1.4"
driver = "cvc4_14"
# --random-seed=42 is not needed as soon as --random-freq=0.0 by default
# to try: --inst-when=full-last-call
# --rlimit=%S : cvc4 1.4 DOES NOT accept -1 as argument
# cvc4 1.4 does not print steps used in --stats anyway
command = "%e --tlimit=%t000 --lang=smt2 %f"
use_at_auto_level = 1

# CVC4 version 1.4, not using SMTLIB bitvectors
[ATP cvc4]
name = "CVC4"
alternative = "noBV"
exec = "cvc4"
exec = "cvc4-1.4"
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
version_old = "1.4"
driver = "cvc4"
# --random-seed=42 is not needed as soon as --random-freq=0.0 by default
# to try: --inst-when=full-last-call
# --rlimit=%S : cvc4 1.4 DOES NOT accept -1 as argument
# cvc4 1.4 does not print steps used in --stats anyway
command = "%e --tlimit=%t000 --lang=smt2 %f"


# CVC4 version 1.0 to 1.3
[ATP cvc4]
name = "CVC4"
exec = "cvc4"
exec = "cvc4-1.3"
exec = "cvc4-1.2"
exec = "cvc4-1.1"
exec = "cvc4-1.0"
version_switch = "--version"
version_regexp = "This is CVC4 version \\([^ \n\r]+\\)"
version_old = "1.3"
version_old = "1.2"
version_old = "1.1"
version_old = "1.0"
driver = "cvc4"
command = "%e --lang=smt2 %f"

# Psyche version 2.x
```

```
[ATP psyche]
name = "Psyche"
exec = "psyche"
exec = "psyche-2.02"
version_switch = "-version"
version_regexp = "\\([^ \n\r]+\\)"
version_ok = "2.0"
driver = "psyche"
command = "%e -gplugin dpll_wl %f"

# CVC3 versions 2.4.x
[ATP cvc3]
name = "CVC3"
exec = "cvc3"
exec = "cvc3-2.4.1"
exec = "cvc3-2.4"
version_switch = "-version"
version_regexp = "This is CVC3 version \\([^ \n]+\\)"
version_ok = "2.4.1"
version_old = "2.4"
# the -timeout option is unreliable in CVC3 2.4.1
command = "%e -seed 42 %f"
driver = "cvc3"

# CVC3 versions 2.x
[ATP cvc3]
name = "CVC3"
exec = "cvc3"
exec = "cvc3-2.2"
exec = "cvc3-2.1"
version_switch = "-version"
version_regexp = "This is CVC3 version \\([^ \n]+\\)"
version_old = "2.2"
version_old = "2.1"
command = "%e -seed 42 -timeout %t %f"
driver = "cvc3"

[ATP yices]
name = "Yices"
exec = "yices"
exec = "yices-1.0.38"
version_switch = "--version"
version_regexp = "\\([^ \n]+\\)"
version_ok = "1.0.38"
version_old = "^1\.0\.3[0-7]$"
version_old = "^1\.0\.2[5-9]$"
version_old = "^1\.0\.2[0-4]$"
version_old = "^1\.0\.1\.*$"
command = "%e"
driver = "yices"

[ATP yices-smt2]
name = "Yices"
exec = "yices-smt2"
exec = "yices-smt2-2.3.0"
version_switch = "--version"
version_regexp = "^Yices \\([^ \n]+\\)$"
version_ok = "2.3.0"
command = "%e"
driver = "yices-smt2"
```

```
[ATP eprover]
name = "Eprover"
exec = "eprover"
exec = "eprover-2.0"
exec = "eprover-1.9.1"
exec = "eprover-1.9"
exec = "eprover-1.8"
exec = "eprover-1.7"
exec = "eprover-1.6"
exec = "eprover-1.5"
exec = "eprover-1.4"
version_switch = "--version"
version_regexp = "E \\([-0-9.]+\\) [^\n]+"
version_ok = "2.0"
version_old = "1.9.1-001"
version_old = "1.9"
version_old = "1.8-001"
version_old = "1.7"
version_old = "1.6"
version_old = "1.5"
version_old = "1.4"
command = "%e -s -R -xAuto -tAuto --cpu-limit=%t --tstp-in %f"
driver = "eprover"
use_at_auto_level = 2

[ATP gappa]
name = "Gappa"
exec = "gappa"
exec = "gappa-1.3.2"
exec = "gappa-1.3.0"
exec = "gappa-1.2.2"
exec = "gappa-1.2.0"
exec = "gappa-1.1.1"
exec = "gappa-1.1.0"
exec = "gappa-1.0.0"
exec = "gappa-0.16.1"
exec = "gappa-0.14.1"
version_switch = "--version"
version_regexp = "Gappa \\([^ \n]*\\)"
version_ok = "^1\.[0-3]\..+$"
version_old = "^0\.1[1-8]\..+$"
command = "%e -Eprecision=70"
driver = "gappa"

[ATP mathsat]
name = "MathSAT5"
exec = "mathsat"
exec = "mathsat-5.2.2"
version_switch = "-version"
version_regexp = "MathSAT5 version \\([^ \n]+\\)"
version_ok = "5.2.2"
command = "%e -input=smt2 -model -random_seed=80"
driver = "mathsat"

[ATP simplify]
name = "Simplify"
exec = "Simplify"
exec = "simplify"
exec = "Simplify-1.5.4"
exec = "Simplify-1.5.5"
version_switch = "-version"
```

```
version_regexp = "Simplify version \\([^ \n,]+\\)"
version_old = "1.5.5"
version_old = "1.5.4"
command = "%e %f"
driver = "simplify"

[ATP metis]
name = "Metis"
exec = "metis"
version_switch = "-v"
version_regexp = "metis \\([^ \n,]+\\)"
version_ok = "2.3"
command = "%e --time-limit %t %f"
driver = "metis"

[ATP metitarski]
name = "MetiTarski"
exec = "metit"
exec = "metit-2.4"
exec = "metit-2.2"
version_switch = "-v"
version_regexp = "MetiTarski \\([^ \n,]+\\)"
version_ok = "2.4"
version_old = "2.2"
command = "%e --time %t %f"
driver = "metitarski"

[ATP polypaver]
name = "PolyPaver"
exec = "polypaver"
exec = "polypaver-0.3"
version_switch = "--version"
version_regexp = "PolyPaver \\([0-9.]+\\) (c)"
version_ok = "0.3"
command = "%e -d 2 -m 10 --time=%t %f"
driver = "polypaver"

[ATP spass]
name = "Spass"
exec = "SPASS"
exec = "SPASS-3.7"
version_switch = " | grep 'SPASS V'"
version_regexp = "SPASS V \\([^ \n\t]+\\)"
version_ok = "3.7"
command = "%e -TPTP -PGiven=0 -PProblem=0 -TimeLimit=%t %f"
driver = "spass"
use_at_auto_level = 2

[ATP spass]
name = "Spass"
exec = "SPASS"
exec = "SPASS-3.8ds"
version_switch = " | grep 'SPASS[^ \\n\\t]* V'"
version_regexp = "SPASS[^ \n\t]* V \\([^ \n\t]+\\)"
version_ok = "3.8ds"
command = "%e -Isabelle=1 -PGiven=0 -TimeLimit=%t %f"
driver = "spass_types"
use_at_auto_level = 2

[ATP vampire]
name = "Vampire"
```

```
exec = "vampire"
exec = "vampire-0.6"
version_switch = "--version"
version_regexp = "Vampire \\([0-9.]+\\)"
command = "%e -t %t"
driver = "vampire"
version_ok = "0.6"

[ATP princess]
name = "Princess"
exec = "princess"
exec = "princess-2015-12-07"
# version_switch = "-h"
# version_regexp = "(CASC version \\([0-9-]+\\))"
version_regexp = "(release \\([0-9-]+\\))"
command = "%e -timeout=%t %f"
driver = "princess"
# version_ok = "2013-05-13"
version_ok = "2015-12-07"

[ATP beagle]
name = "Beagle"
exec = "beagle"
exec = "beagle-0.4.1"
# version_switch = "-h"
version_regexp = "version \\([0-9.]+\\)"
command = "%e %f"
driver = "beagle"
version_ok = "0.4.1"

[ATP verit]
name = "veriT"
exec = "veriT"
exec = "veriT-201410"
version_switch = "--version"
version_regexp = "version \\([^ \n\r]+\\)"
command = "%e --disable-print-success %f"
driver = "verit"
version_ok = "201410"

[ATP verit]
name = "veriT"
exec = "veriT"
exec = "veriT-201310"
version_switch = "--version"
version_regexp = "version \\([^ \n\r]+\\)"
command = "%e --disable-print-success --enable-simp \
--enable-unit-simp --enable-simp-sym --enable-unit-subst-simp --enable-bclause %f"
driver = "verit"
version_old = "201310"

# Z3 >= 4.6.0, with counterexamples and incremental usage
[ATP z3-ce]
name = "Z3"
alternative = "counterexamples"
exec = "z3"
exec = "z3-4.8.6"
exec = "z3-4.8.5"
exec = "z3-4.8.4"
exec = "z3-4.8.3"
exec = "z3-4.8.1"
```

```
exec = "z3-4.7.1"
exec = "z3-4.6.0"
exec = "z3-4.5.0"
exec = "z3-4.4.1"
exec = "z3-4.4.0"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_ok = "4.8.6"
version_ok = "4.8.5"
version_ok = "4.8.4"
version_ok = "4.8.3"
version_ok = "4.8.1"
version_ok = "4.7.1"
version_ok = "4.6.0"
version_ok = "4.5.0"
version_old = "4.4.1"
version_old = "4.4.0"
driver = "z3_440_counterexample"
# -t sets the time limit per query
command = "%e -smt2 -t:%t000 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 %f"
command_steps = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 memory_max_alloc_count=

# Z3 >= 4.4.0, with BV support
[ATP z3]
name = "Z3"
exec = "z3"
exec = "z3-4.8.6"
exec = "z3-4.8.5"
exec = "z3-4.8.4"
exec = "z3-4.8.3"
exec = "z3-4.8.1"
exec = "z3-4.7.1"
exec = "z3-4.6.0"
exec = "z3-4.5.0"
exec = "z3-4.4.1"
exec = "z3-4.4.0"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_ok = "4.8.6"
version_ok = "4.8.5"
version_ok = "4.8.4"
version_ok = "4.8.3"
version_ok = "4.8.1"
version_ok = "4.7.1"
version_ok = "4.6.0"
version_ok = "4.5.0"
version_old = "4.4.1"
version_old = "4.4.0"
driver = "z3_440"
command = "%e -smt2 -T:%t sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 %f"
command_steps = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 memory_max_alloc_count=
use_at_auto_level = 1

# Z3 >= 4.4.0, without BV support
[ATP z3-nobv]
name = "Z3"
alternative = "noBV"
exec = "z3"
exec = "z3-4.8.6"
exec = "z3-4.8.5"
exec = "z3-4.8.4"
```

```
exec = "z3-4.8.3"
exec = "z3-4.8.1"
exec = "z3-4.7.1"
exec = "z3-4.6.0"
exec = "z3-4.5.0"
exec = "z3-4.4.1"
exec = "z3-4.4.0"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_ok = "4.8.6"
version_ok = "4.8.5"
version_ok = "4.8.4"
version_ok = "4.8.3"
version_ok = "4.8.1"
version_ok = "4.7.1"
version_ok = "4.6.0"
version_ok = "4.5.0"
version_old = "4.4.1"
version_old = "4.4.0"
driver = "z3_432"
command = "%e -smt2 -T:%t sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 %f"
command_steps = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 memory_max_alloc_count=

# Z3 4.3.2 does not support option global option -rs anymore.
# use settings given by "z3 -p" instead
# Z3 4.3.2 supports Datatypes
[ATP z3]
name = "Z3"
exec = "z3-4.3.2"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_old = "4.3.2"
driver = "z3_432"
command = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 %f"
command_steps = "%e -smt2 sat.random_seed=42 nlsat.randomize=false smt.random_seed=42 memory_max_alloc_count=

[ATP z3]
name = "Z3"
exec = "z3"
exec = "z3-4.3.1"
exec = "z3-4.3.0"
exec = "z3-4.2"
exec = "z3-4.1.2"
exec = "z3-4.1.1"
exec = "z3-4.0"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_old = "4.3.1"
version_old = "4.3.0"
version_old = "4.2"
version_old = "4.1.2"
version_old = "4.1.1"
version_old = "4.0"
driver = "z3"
command = "%e -smt2 -rs:42 %f"

[ATP z3]
name = "Z3"
exec = "z3"
exec = "z3-3.2"
exec = "z3-3.1"
```

```
exec = "z3-3.0"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_old = "3.2"
version_old = "3.1"
version_old = "3.0"
driver = "z3"
# the -T is unreliable in Z3 3.2
command = "%e -smt2 -rs:42 %f"

[ATP z3]
name = "Z3"
exec = "z3"
exec = "z3-2.19"
exec = "z3-2.18"
exec = "z3-2.17"
exec = "z3-2.16"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_old = "^2\.2.+$"
version_old = "^2\.1[6-9]$"
driver = "z3"
command = "%e -smt2 -rs:42 \
PHASE_SELECTION=0 \
RESTART_STRATEGY=0 \
RESTART_FACTOR=1.5 \
QI_EAGER_THRESHOLD=100 \
ARITH_RANDOM_INITIAL_VALUE=true \
SORT_AND_OR=false \
CASE_SPLIT=3 \
DELAY_UNITS=true \
DELAY_UNITS_THRESHOLD=16 \
%f"
#Other Parameters given by Nikolaj Bjorner
#BV_REFLECT=true #arith?
#MODEL_PARTIAL=true
#MODEL_VALUE_COMPLETION=false
#MODEL_HIDE_UNUSED_PARTITIONS=false
#MODEL_V1=true
#ASYNC_COMMANDS=false
#NNF_SK_HACK=true

[ATP z3]
name = "Z3"
exec = "z3"
exec = "z3-2.2"
exec = "z3-2.1"
exec = "z3-1.3"
version_switch = "-version"
version_regexp = "Z3 version \\([^ \n\r]+\\)"
version_old = "^2\.1[0-5]$"
version_old = "^2\.[0-9]$"
version_old = "1.3"
command = "%e -smt %f"
driver = "z3_smtv1"

[ATP zenon]
name = "Zenon"
exec = "zenon"
exec = "zenon-0.8.0"
exec = "zenon-0.7.1"
```

```
version_switch = "-v"
version_regexp = "zenon version \\([^ \n\t]+\\)"
version_ok = "0.8.0"
version_ok = "0.7.1"
command = "%e -p0 -itptp -max-size %mM -max-time %ts %f"
driver = "zenon"

[ATP zenon_modulo]
name = "Zenon Modulo"
exec = "zenon_modulo"
version_switch = "-v"
version_regexp = "zenon_modulo version \\([0-9.]+\\)"
version_ok = "0.4.1"
command = "%e -p0 -itptp -max-size %mM -max-time %ts %f"
driver = "zenon_modulo"

[ATP iprover]
name = "iProver"
exec = "iprover"
exec = "iprover-0.8.1"
version_switch = " | grep iProver"
version_regexp = "iProver v\\([^ \n\t]+\\)"
version_ok = "0.8.1"
command = "%e --fof true --out_options none \
--time_out_virtual %t --clausifier /usr/bin/env --clausifier_options \
\"eprover --cnf --tstp-format \" %f"
driver = "iprover"

[ATP mathematica]
name = "Mathematica"
exec = "math"
version_switch = "-run \"Exit[]\""
version_regexp = "Mathematica \\([0-9.]+\\)"
version_ok = "9.0"
version_ok = "8.0"
version_ok = "7.0"
command = "%e -noprompt"
driver = "mathematica"

[ITP coq]
name = "Coq"
support_library = "%l/coq/version"
exec = "coqtop"
version_switch = "-v"
version_regexp = "The Coq Proof Assistant, version \\([^ \n]+\\)"
version_ok = "^8\.9\.[0-1]$"
version_ok = "^8\.8\.[0-2]$"
version_ok = "^8\.7\.[0-2]$"
version_ok = "8.6.1"
version_ok = "8.6"
version_old = "^8\.5pl[1-3]$"
version_old = "8.5"
command = "%e -batch -R %l/coq Why3 -l %f"
driver = "coq"
editor = "coqide"

[ITP pvs]
name = "PVS"
support_library = "%l/pvs/version"
exec = "pvs"
version_switch = "-version"
```

```
version_regexp = "PVS Version \\([^ \n]+\\)"
version_ok = "6.0"
version_bad = "^[0-5]\..+$"
command = "%l/why3-call-pvs %l/pvs proveit -f %f"
driver = "pvs"
in_place = true
editor = "pvs"

[ITP isabelle]
name = "Isabelle"
exec = "isabelle"
version_switch = "version"
version_regexp = "Isabelle\\([0-9]+\\(-[0-9]+\\)?\\)"
version_ok = "2018"
version_bad = "2017"
version_bad = "2016-1"
command = "%e why3 -b %f"
driver = "isabelle2018"
in_place = true
editor = "isabelle-jedit"

[ITP isabelle]
name = "Isabelle"
exec = "isabelle"
version_switch = "version"
version_regexp = "Isabelle\\([0-9]+\\(-[0-9]+\\)?\\)"
version_ok = "2017"
version_bad = "2018"
version_bad = "2016-1"
command = "%e why3 -b %f"
driver = "isabelle2017"
in_place = true
editor = "isabelle-jedit"

[editor pvs]
name = "PVS"
command = "%l/why3-call-pvs %l pvs %f"

[editor coqide]
name = "CoqIDE"
command = "coqide -R %l/coq Why3 %f"

[editor proofgeneral-coq]
name = "Emacs/ProofGeneral/Coq"
command = "emacs --eval \"(setq coq-load-path '((\\\"%l/coq\\\" \\\"Why3\\\")))\" %f"

[editor isabelle-jedit]
name = "Isabelle/jEdit"
command = "isabelle why3 -i %f"

[editor altgr-ergo]
name = "AltGr-Ergo"
command = "altgr-ergo %f"

[shortcut shortcut1]
name="Alt-Ergo"
shortcut="altergo"
```

9.3 The `why3.conf` Configuration File

One can use a custom configuration file. The Why3 tools look for it in the following order:

1. the file specified by the `-C` or `--config` options,
2. the file specified by the environment variable `WHY3CONFIG` if set,
3. the file `$HOME/.why3.conf` (`$USERPROFILE/.why3.conf` under Windows) or, in the case of local installation, `why3.conf` in the top directory of Why3 sources.

If none of these files exist, a built-in default configuration is used.

A section begins with a header inside square brackets and ends at the beginning of the next section. The header of a section can be a single identifier, *e.g.* `[main]`, or it can be composed by a family name and a single family argument, *e.g.* `[prover alt-ergo]`.

Sections contain associations `key=value`. A value is either an integer (*e.g.* `-555`), a boolean (`true`, `false`), or a string (*e.g.* `"emacs"`). Some specific keys can be attributed multiple values and are thus allowed to occur several times inside a given section. In that case, the relative order of these associations matters.

9.4 Drivers for External Provers

Drivers for external provers are readable files from directory `drivers`. Experimented users can modify them to change the way the external provers are called, in particular which transformations are applied to goals.

[TO BE COMPLETED LATER]

9.5 Transformations

This section documents the available transformations. We first describe the most important ones, and then we provide a quick documentation of the others, first the non-splitting ones, *e.g.* those which produce exactly one goal as result, and the others which produce any number of goals.

Notice that the set of available transformations in your own installation is given by

```
why3 --list-transforms
```

9.5.1 Inlining definitions

Those transformations generally amount to replace some applications of function or predicate symbols with its definition.

inline_trivial expands and removes definitions of the form

```
function  f x_1 ... x_n = (g e_1 ... e_k)
predicate p x_1 ... x_n = (q e_1 ... e_k)
```

when each e_i is either a ground term or one of the x_j, and each $x_1 \dots x_n$ occurs at most once in all the e_i.

inline_goal expands all outermost symbols of the goal that have a non-recursive definition.

inline_all expands all non-recursive definitions.

9.5.2 Induction Transformations

induction_ty_lex performs structural, lexicographic induction on goals involving universally quantified variables of algebraic data types, such as lists, trees, etc. For instance, it transforms the following goal

```
goal G: forall l: list 'a. length l >= 0
```

into this one:

```
goal G :
  forall l:list 'a.
     match l with
     | Nil -> length l >= 0
     | Cons a l1 -> length l1 >= 0 -> length l >= 0
     end
```

When induction can be applied to several variables, the transformation picks one heuristically. The `[@induction]` attribute can be used to force induction over one particular variable, *e.g.* with

```
goal G: forall l1 [@induction] l2 l3: list 'a.
        l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3
```

induction will be applied on `l1`. If this attribute is attached to several variables, a lexicographic induction is performed on these variables, from left to right.

9.5.3 Simplification by Computation

These transformations simplify the goal by applying several kinds of simplification, described below. The transformations differ only by the kind of rules they apply:

compute_in_goal aggressively applies all known computation/simplification rules.

compute_specified performs rewriting using only built-in operators and user-provided rules.

The kinds of simplification are as follows.

- Computations with built-in symbols, *e.g.* operations on integers, when applied to explicit constants, are evaluated. This includes evaluation of equality when a decision can be made (on integer constants, on constructors of algebraic data types), Boolean evaluation, simplification of pattern-matching/conditional expression, extraction of record fields, and beta-reduction. At best, these computations reduce the goal to `true` and the transformations thus does not produce any sub-goal. For example, a goal like `6*7=42` is solved by those transformations.
- Unfolding of definitions, as done by `inline_goal`. Transformation `compute_in_goal` unfolds all definitions, including recursive ones. For `compute_specified`, the user can enable unfolding of a specific logic symbol by attaching the meta `rewrite_def` to the symbol.

```
function sqr (x:int) : int = x * x
meta "rewrite_def" function sqr
```

- Rewriting using axioms or lemmas declared as rewrite rules. When an axiom (or a lemma) has one of the forms

```
axiom a: forall ... t1 = t2
```

or

```
axiom a: forall ... f1 <-> f2
```

then the user can declare

```
meta "rewrite" prop a
```

to turn this axiom into a rewrite rule. Rewriting is always done from left to right. Beware that there is no check for termination nor for confluence of the set of rewrite rules declared.

Instead of using a meta, it is possible to declare an axiom as a rewrite rule by adding the `[@rewrite]` attribute on the axiom name or on the axiom itself, e.g.:

```
axiom a [@rewrite]: forall ... t1 = t2
lemma b: [@rewrite] forall ... f1 <-> f2
```

The second form allows some form of local rewriting, e.g.

```
lemma l: forall x y. ([@rewrite] x = y) -> f x = f y
```

can be proved by `introduce_premises` followed by `compute_specified`.

Bound on the number of reductions The computations performed by these transformations can take an arbitrarily large number of steps, or even not terminate. For this reason, the number of steps is bounded by a maximal value, which is set by default to 1000. This value can be increased by another meta, *e.g.*

```
meta "compute_max_steps" 1_000_000
```

When this upper limit is reached, a warning is issued, and the partly-reduced goal is returned as the result of the transformation.

9.5.4 Other Non-Splitting Transformations

eliminate_algebraic replaces algebraic data types by first-order definitions [10].

eliminate_builtin removes definitions of symbols that are declared as builtin in the driver, *i.e.* with a "syntax" rule.

eliminate_definition_func replaces all function definitions with axioms.

eliminate_definition_pred replaces all predicate definitions with axioms.

eliminate_definition applies both transformations above.

eliminate_mutual_recursion replaces mutually recursive definitions with axioms.

eliminate_recursion replaces all recursive definitions with axioms.

eliminate_if_term replaces terms of the form `if formula then t2 else t3` by lifting them at the level of formulas. This may introduce `if then else` in formulas.

eliminate_if_fmla replaces formulas of the form `if f1 then f2 else f3` by an equivalent formula using implications and other connectives.

eliminate_if applies both transformations above.

eliminate_inductive replaces inductive predicates by (incomplete) axiomatic definitions, *i.e.* construction axioms and an inversion axiom.

eliminate_let_fmla eliminates `let` by substitution, at the predicate level.

eliminate_let_term eliminates `let` by substitution, at the term level.

eliminate_let applies both transformations above.

encoding_smt encodes polymorphic types into monomorphic types [3].

encoding_tptp encodes theories into unsorted logic.

introduce_premises moves antecedents of implications and universal quantifications of the goal into the premises of the task.

simplify_array automatically rewrites the task using the lemma `Select_eq` of theory `map.Map`.

simplify_formula reduces trivial equalities $t = t$ to true and then simplifies propositional structure: removes true, false, simplifies $f \wedge f$ to f, etc.

simplify_recursive_definition reduces mutually recursive definitions if they are not really mutually recursive, *e.g.*

```
function f : ... = ... g ...
with g : ... = e
```

becomes

```
function g : ... = e
function f : ... = ... g ...
```

if f does not occur in e.

simplify_trivial_quantification simplifies quantifications of the form

```
forall x, x = t -> P(x)
```

into

```
P(t)
```

when x does not occur in t. More generally, this simplification is applied whenever $x = t$ or $t = x$ appears in negative position.

simplify_trivial_quantification_in_goal is the same as above but it applies only in the goal.

split_premise replaces axioms in conjunctive form by an equivalent collection of axioms. In absence of case analysis attributes (see `split_goal` for details), the number of axiom generated per initial axiom is linear in the size of that initial axiom.

split_premise_full is similar to `split_premise`, but it also converts the axioms to conjunctive normal form. The number of axioms generated per initial axiom may be exponential in the size of the initial axiom.

9.5.5 Other Splitting Transformations

simplify_formula_and_task is the same as `simplify_formula` but it also removes the goal if it is equivalent to true.

split_goal changes conjunctive goals into the corresponding set of subgoals. In absence of case analysis attributes, the number of subgoals generated is linear in the size of the initial goal.

Behavior on asymmetric connectives and `by`/`so` The transformation treats specially asymmetric and `by`/`so` connectives. Asymmetric conjunction `A && B` in goal position is handled as syntactic sugar for `A /\ (A -> B)`. The conclusion of the first subgoal can then be used to prove the second one.

Asymmetric disjunction `A || B` in hypothesis position is handled as syntactic sugar for `A \/ ((not A) /\ B)`. In particular, a case analysis on such hypothesis would give the negation of the first hypothesis in the second case.

The `by` connective is treated as a proof indication. In hypothesis position, `A by B` is treated as if it were syntactic sugar for its regular interpretation `A`. In goal position, it is treated as if `B` was an intermediate step for proving `A`. `A by B` is then replaced by `B` and the transformation also generates a side-condition subgoal `B -> A` representing the logical cut.

Although splitting stops at disjunctive points like symmetric disjunction and left-hand sides of implications, the occurrences of the `by` connective are not restricted. For instance:

- Splitting

```
goal G : (A by B) && C
```

  generates the subgoals

```
goal G1 : B
goal G2 : A -> C
goal G3 : B -> A (* side-condition *)
```

- Splitting

```
goal G : (A by B) \/ (C by D)
```

  generates

```
goal G1 : B \/ D
goal G2 : B -> A (* side-condition *)
goal G3 : D -> C (* side-condition *)
```

- Splitting

```
goal G : (A by B) || (C by D)
```

generates

```
goal G1 : B || D
goal G2 : B -> A          (* side-condition *)
goal G3 : B || (D -> C)   (* side-condition *)
```

Note that due to the asymmetric disjunction, the disjunction is kept in the second side-condition subgoal.

- Splitting

```
goal G : exists x. P x by x = 42
```

generates

```
goal G1 : exists x. x = 42
goal G2 : forall x. x = 42 -> P x (* side-condition *)
```

Note that in the side-condition subgoal, the context is universally closed.

The `so` connective plays a similar role in hypothesis position, as it serves as a consequence indication. In goal position, `A so B` is treated as if it were syntactic sugar for its regular interpretation `A`. In hypothesis position, it is treated as if both `A` and `B` were true because `B` is a consequence of `A`. `A so B` is replaced by `A /\ B` and the transformation also generates a side-condition subgoal `A -> B` corresponding to the consequence relation between formula.

As with the `by` connective, occurrences of `so` are unrestricted. For instance:

- Splitting

```
goal G : (((A so B) \/ C) -> D) && E
```

generates

```
goal G1 : ((A /\ B) \/ C) -> D
goal G2 : (A \/ C -> D) -> E
goal G3 : A -> B                (* side-condition *)
```

- Splitting

```
goal G : A by exists x. P x so Q x so R x by T x
(* reads: A by (exists x. P x so (Q x so (R x by T x))) *)
```

generates

```
goal G1 : exists x. P x
goal G2 : forall x. P x -> Q x                (* side-condition *)
goal G3 : forall x. P x -> Q x -> T x         (* side-condition *)
goal G4 : forall x. P x -> Q x -> T x -> R x  (* side-condition *)
goal G5 : (exists x. P x /\ Q x /\ R x) -> A  (* side-condition *)
```

In natural language, this corresponds to the following proof scheme for `A`: There exists a `x` for which `P` holds. Then, for that witness `Q` and `R` also holds. The last one holds because `T` holds as well. And from those three conditions on `x`, we can deduce `A`.

Attributes controlling the transformation The transformations in the split family can be controlled by using attributes on formulas.

The `[@stop_split]` attribute can be used to block the splitting of a formula. The attribute is removed after blocking, so applying the transformation a second time will split the formula. This is can be used to decompose the splitting process in several steps. Also, if a formula with this attribute is found in non-goal position, its `by`/`so` proof indication will be erased by the transformation. In a sense, formulas tagged by `[@stop_split]` are handled as if they were local lemmas.

The `[@case_split]` attribute can be used to force case analysis on hypotheses. For instance, applying `split_goal` on

```
goal G : ([@case_split] A \/ B) -> C
```

generates the subgoals

```
goal G1 : A -> C
goal G2 : B -> C
```

Without the attribute, the transformation does nothing because undesired case analysis may easily lead to an exponential blow-up.

Note that the precise behavior of splitting transformations in presence of the `[@case_split]` attribute is not yet specified and is likely to change in future versions.

split_all performs both `split_premise` and `split_goal`.

split_intro performs both `split_goal` and `introduce_premises`.

split_goal_full has a behavior similar to `split_goal`, but also converts the goal to conjunctive normal form. The number of subgoals generated may be exponential in the size of the initial goal.

split_all_full performs both `split_premise` and `split_goal_full`.

9.6 Proof Strategies

As seen in Section 5.3, the IDE provides a few buttons that trigger the run of simple proof strategies on the selected goals. Proof strategies can be defined using a basic assembly-style language, and put into the Why3 configuration file. The commands of this basic language are:

- `c` *p t m* calls the prover *p* with a time limit *t* and memory limit *m*. On success, the strategy ends, it continues to next line otherwise
- `t` *n lab* applies the transformation *n*. On success, the strategy continues to label *lab*, and is applied to each generated sub-goals. It continues to next line otherwise.
- `g` *lab* inconditionally jumps to label *lab*
- *lab*`:` declares the label *lab*. The default label `exit` allows to stop the program.

To examplify this basic programming language, we give below the default strategies that are attached to the default buttons of the IDE, assuming that the provers Alt-Ergo 1.30, CVC4 1.5 and Z3 4.5.0 were detected by the `why3 config --detect` command

Split is bound to the 1-line strategy

```
t split_goal_wp exit
```

Auto level 0 is bound to

```
c Z3,4.5.0, 1 1000
c Alt-Ergo,1.30, 1 1000
c CVC4,1.5, 1 1000
```

The three provers are tried for a time limit of 1 second and memory limit of 1 Gb, each in turn. This is a perfect strategy for a first attempt to discharge a new goal.

Auto level 1 is bound to

```
start:
c Z3,4.5.0, 1 1000
c Alt-Ergo,1.30, 1 1000
c CVC4,1.5, 1 1000
t split_goal_wp start
c Z3,4.5.0, 10 4000
c Alt-Ergo,1.30, 10 4000
c CVC4,1.5, 10 4000
```

The three provers are first tried for a time limit of 1 second and memory limit of 1 Gb, each in turn. If none of them succeed, a split is attempted. If the split works then the same strategy is retried on each sub-goals. If the split does not succeed, the provers are tried again with a larger limits.

Auto level 2 is bound to

```
start:
c Z3,4.5.0, 1 1000
c Eprover,2.0, 1 1000
c Spass,3.7, 1 1000
c Alt-Ergo,1.30, 1 1000
c CVC4,1.5, 1 1000
t split_goal_wp start
c Z3,4.5.0, 5 2000
c Eprover,2.0, 5 2000
c Spass,3.7, 5 2000
c Alt-Ergo,1.30, 5 2000
c CVC4,1.5, 5 2000
t introduce_premises afterintro
afterintro:
t inline_goal afterinline
g trylongertime
afterinline:
t split_goal_wp start
trylongertime:
c Z3,4.5.0, 30 4000
```

```
c Eprover,2.0, 30 4000
c Spass,3.7, 30 4000
c Alt-Ergo,1.30, 30 4000
c CVC4,1.5, 30 4000
```

Notice that now 5 provers are used. The provers are first tried for a time limit of 1 second and memory limit of 1 Gb, each in turn. If none of them succeed, a split is attempted. If the split works then the same strategy is retried on each sub-goals. If the split does not succeed, the prover are tried again with limits of 5 s and 2 Gb. If all fail, we attempt the transformation of introduction of premises in the context, followed by an inlining of the definitions in the goals. We then attempt a split again, if the split succeeds, we restart from the beginning, if it fails then provers are tried again with 30s and 4 Gb.

Part III

Appendix

Appendix A

Release Notes

A.1 Release Note for version 1.2: new syntax for "auto-dereference"

Version 1.2 introduces a simplified syntax for reference variables, to avoid the somehow heavy OCaml syntax using bang character. In short, this is syntactic sugar summarized in the following table. An example using this new syntax is in `examples/isqrt.mlw`

auto-dereference syntax	**desugared to**
`let &x = ... in`	`let (x: ref ...) = ... in`
`f x;`	`f x.contents;`
`x <- ...`	`x.contents <- ...`
`let ref x = ...`	`let &x = ref ...`

Notice that:

- The `&` marker adds the typing constraint `(x: ref ...)`
- Top-level `let/val ref` and `let/val &` are allowed
- Auto-dereferencing works in logic, but such variables cannot be introduced inside logical terms.

That syntactic sugar is further extended to pattern matching, function parameters and reference passing as follows.

auto-dereference syntax	**desugared to**
`match e with (x,&y) -> y end`	`match e with (x,(y: ref ...)) -> y.contents end`

`let incr (&x: ref int) =`
`  x <- x + 1`
 	
`let f () =`
`  let ref x = 0 in`
`  incr x;`
`  x`
	`let incr (x: ref int) =`
	`  x.contents <- x.contents + 1`
	
	`let f () =`
	`  let x = ref 0 in`
	`  incr x;`
	`  x.contents`

`let incr (ref x: int) ...`	`let incr (&x: ref int) ...`

The type annotation is not required. Let-functions with such formal parameters also prevent the actual argument from auto-dereferencing when used in logic. Pure logical symbols cannot be declared with such parameters.

Auto-dereference suppression does not work in the middle of a relation chain: in `0 < x :< 17`, `x` will be dereferenced even if `(:<)` expects a ref-parameter on the left.

Finally, that syntactic sugar applies to the caller side:

auto-dereference syntax	**desugared to**

`let f () =`
`  let ref x = 0 in`
`  g &x`
	`let f () =`
	`  let x = ref 0 in`
	`  g x`

The `&` marker can only be attached to a variable. Works in logic.

Ref-binders and `&`-binders in variable declarations, patterns, and function parameters do not require importing `ref.Ref`. Any example that does not use references inside data structures can be rewritten by using ref-binders, without importing `ref.Ref`.

Explicit use of type symbol "ref", program function "ref", or field "contents" require importing `ref.Ref` or `why3.Ref.Ref`. Operations `(:=)` and `(!)` require importing `ref.Ref`.

Operation `(:=)` is fully subsumed by direct assignment `(<-)`.

A.2 Release Notes for version 1.0: syntax changes w.r.t. 0.88

The syntax of WhyML programs changed in release 1.0. The table in Figure A.1 summarizes the changes.

Note also that logical symbols can no longer be used in non-ghost code; in particular, there is no polymorphic equality in programs anymore, so equality functions must be declared/defined on a per-type basis (already done for type `int` in the standard library). The `CHANGES.md` file describes other potential sources of incompatibility.

Here are a few more semantic changes

Proving only partial correctness In versions 0.xx of Why3, when a program function is recursive but not given a variant, or contains a while loop not given a variant, then it was assumed that the user wants to prove partial correctness only. A warning was issued, recommending to add an extra `diverges` clause to that function's contract. It was also possible to disable that warning by adding the label `"W:diverges:N"` to the function's name. Version 1.00 of Why3 is more aggressively requiring the user to prove the termination of functions which are not given the `diverges` clause, and the previous warning is now an error. The possibility of proving only partial correctness is now given on a more fine-grain basis: any expression for which one wants to prove partial correctness only, must by annotated with the attribute `[@vc:divergent]`.

In other words, if one was using the following structure in Why3 0.xx:

```
let f "W:diverges:N" <parameters> <contract> = <body>
```

then in 1.00 it should be written as

```
let f <parameters> <contract> = [@vc:divergent] <body>
```

version 0.88	**version 1.0**
`function f ...`	`let function f ...` if called in programs
`'L:`	`label L in`
`at x 'L`	`x at L`
`\x. e`	`fun x -> e`
`use HighOrd`	nothing, not needed anymore
`HighOrd.pred ty`	`ty -> bool`
`type t model ...`	`type t = abstract ...`
`abstract e ensures { Q }`	`begin ensures { Q } e end`
`namespace N`	`scope N`
`use import M`	`use M`
`"attribute"`	`[@attribute]`
`meta M prop P`	`meta M lemma P` or `meta M axiom P` or `meta M goal P`
`loop ... end`	`while true do ... done`
`type ... invariant { ... self.foo ... }`	`type ... invariant { ... foo ... }`

Figure A.1: Syntax changes from version 0.88 to version 1.0

Semantics of the `any` construct The `any` construct in Why3 0.xx was introducing an arbitrary value satisfying the associated post-condition. In some sense, this construct was introducing some form of an axiom stating that such a value exists. In Why3 1.00, it is now mandatory to prove the existence of such a value, and a VC is generated for that purpose.

To obtain the effect of the former semantics of the `any` construct, one should use instead a local `val` function. In other words, if one was using the following structure in Why3 0.xx:

```
any t ensures { P }
```

then in 1.00 it should be written as

```
val x:t ensures { P } in x
```

A.3 Release Notes for version 0.80: syntax changes w.r.t. 0.73

The syntax of WhyML programs changed in release 0.80. The table in Figure A.2 summarizes the changes.

A.4 Summary of Changes w.r.t. Why 2

The main new features with respect to Why 2.xx are the following.

1. Completely redesigned input syntax for logic declarations
   - new syntax for terms and formulas

version 0.73	**version 0.80**
`type t = {| field : int |}`	`type t = { field : int }`
`{| field = 5 |}`	`{ field = 5 }`
`use import module M`	`use import M`

`let rec f (x:int) (y:int) : t`
`  variant { t } with rel =`
`  { P }`
`  e`
`  { Q }`
`  | Exc1 -> { R1 }`
`  | Exc2 n -> { R2 }`
	`let rec f (x:int) (y:int) : t`
	`  variant { t with rel }`
	`  requires { P }`
	`  ensures { Q }`
	`  raises { Exc1 -> R1`
	`  | Exc2 n -> R2 }`
	`  = e`

`val f (x:int) (y:int) :`
`  { P }`
`  t`
`  writes a b`
`  { Q }`
`  | Exc1 -> { R1 }`
`  | Exc2 n -> { R2 }`
	`val f (x:int) (y:int) : t`
	`  requires { P }`
	`  writes { a, b }`
	`  ensures { Q }`
	`  raises { Exc1 -> R1`
	`  | Exc2 n -> R2 }`

`val f : x:int -> y:int ->`
`  { P }`
`  t`
`  writes a b`
`  { Q }`
`  | Exc1 -> { R1 }`
`  | Exc2 n -> { R2 }`
	`val f (x y:int) : t`
	`  requires { P }`
	`  writes { a, b }`
	`  ensures { Q }`
	`  raises { Exc1 -> R1`
	`  | Exc2 n -> R2 }`

`abstract e { Q }`	`abstract e ensures { Q }`

Figure A.2: Syntax changes from version 0.73 to version 0.80

- enumerated and algebraic data types, pattern matching
- recursive definitions of logic functions and predicates, with termination checking
- inductive definitions of predicates
- declarations are structured in components called "theories", which can be reused and instantiated

2. More generic handling of goals and lemmas to prove
   - concept of proof task
   - generic concept of task transformation
   - generic approach for communicating with external provers
3. Source code organized as a library with a documented API, to allow access to Why3 features programmatically.
4. GUI with new features with respect to the former GWhy
   - session save and restore
   - prover calls in parallel
   - splitting, and more generally applying task transformations, on demand

- ability to edit proofs for interactive provers (Coq only for the moment) on any subtask

5. Extensible architecture via plugins
   - users can define new transformations
   - users can add connections to additional provers

Bibliography

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[3] François Bobot, Sylvain Conchon, Évelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In Clark Barrett and Leonardo de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, volume 367 of *ACM International Conference Proceedings Series*, pages 1–5, 2008.

[4] Sylvain Conchon and Évelyne Contejean. The Alt-Ergo automatic theorem prover. http://alt-ergo.lri.fr/, 2008. APP deposit under the number IDDN FR 001 110026 000 S P 2010 000 1000.

[5] Sylvain Dailler, David Hauzar, Claude Marché, and Yannick Moy. Instrumenting a weakest precondition calculus for counterexample generation. *Journal of Logical and Algebraic Methods in Programming*, 99:97–113, 2018.

[6] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

[7] David Hauzar, Claude Marché, and Yannick Moy. Counterexamples from proof failures in SPARK. In Rocco De Nicola and Eva Kühn, editors, *Software Engineering and Formal Methods*, Lecture Notes in Computer Science, pages 215–233, Vienna, Austria, 2016.

[8] IEEE standard for floating-point arithmetic, 2008. https://dx.doi.org/10.1109/IEEESTD.2008.4610935.

[9] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[10] Andrei Paskevich. Algebraic types and pattern matching in the logical language of the Why verification platform. Technical Report 7128, INRIA, 2009. http://hal.inria.fr/inria-00439232.

[11] Natarajan Shankar and Peter Mueller. Verified Software: Theories, Tools and Experiments (VSTTE'10). Software Verification Competition, August 2010. http://www.macs.hw.ac.uk/vstte10/Competition.html.

List of Figures

Index