Why3 Documentation

Release 1.4.1

The Why3 Development Team

Mar 25, 2022

CONTENTS

1 Foreword **3**
- 1.1 Availability . . . 3
- 1.2 Contact . . . 3
- 1.3 Acknowledgements . . . 4

2 Getting Started **5**
- 2.1 Hello Proofs . . . 5
- 2.2 Getting Started with the GUI . . . 6
  - 2.2.1 Calling provers on goals . . . 7
  - 2.2.2 Applying transformations . . . 8
  - 2.2.3 Modifying the input . . . 8
  - 2.2.4 Replaying obsolete proofs . . . 10
  - 2.2.5 Cleaning . . . 10
- 2.3 Getting Started with the Why3 Command . . . 10

3 Why3 by Examples **13**
- 3.1 Problem 0: Einstein's Problem . . . 14
- 3.2 Problem 1: Sum and Maximum . . . 16
- 3.3 Problem 2: Inverting an Injection . . . 18
- 3.4 Problem 3: Searching a Linked List . . . 20
- 3.5 Problem 4: N-Queens . . . 23
- 3.6 Problem 5: Amortized Queue . . . 27

4 The Why3 API **31**
- 4.1 Building Propositional Formulas . . . 31
- 4.2 Building Tasks . . . 32
- 4.3 Calling External Provers . . . 33
- 4.4 Building Terms . . . 36
- 4.5 Building Quantified Formulas . . . 37
- 4.6 Building Theories . . . 37
- 4.7 Operations on Terms and Formulas, Transformations . . . 39
- 4.8 Proof Sessions . . . 40
- 4.9 ML Programs . . . 40
  - 4.9.1 Untyped syntax tree . . . 40
  - 4.9.2 Alternative, top-down, construction of parsing trees . . . 44
  - 4.9.3 Using the parsing trees . . . 45
  - 4.9.4 Use attributes to infer loop invariants . . . 47
  - 4.9.5 Typed declaration . . . 49
- 4.10 Generating counterexamples . . . 52
  - 4.10.1 Attributes and locations on identifiers . . . 52

4.10.2 Attributes in formulas . . . 52
4.10.3 Counterexamples output formats . . . 53

5 Compilation, Installation **55**
5.1 Installing Why3 . . . 55
5.1.1 Installation via Opam . . . 55
5.1.2 Installation via Docker . . . 55
5.1.3 Installation from Source Distribution . . . 56
5.2 Installing External Provers . . . 57
5.2.1 Multiple Versions of the Same Prover . . . 58
5.2.2 Session Update after Prover Upgrade . . . 58
5.3 Configure Editors for editing WhyML sources . . . 58
5.3.1 Emacs . . . 59
5.3.2 Vim . . . 59
5.4 Configure Shells for auto-completion of Why3 command arguments . . . 59
5.5 Inference of Loop Invariants . . . 59

6 Reference Manuals for the Why3 Tools **61**
6.1 The `config` Command . . . 62
6.1.1 Command `add-prover` . . . 62
6.1.2 Command `detect` . . . 63
6.1.3 Command `list-supported-provers` . . . 63
6.1.4 Command `show` . . . 63
6.2 The `prove` Command . . . 63
6.2.1 Prover Results . . . 64
6.2.2 Options . . . 64
6.2.3 Generating potential counterexamples . . . 65
6.2.4 Generating validated counterexamples . . . 65
6.3 The `ide` Command . . . 65
6.3.1 Session . . . 66
6.3.2 Context Menu . . . 66
6.3.3 Global Menus . . . 67
6.3.4 Command-line interface . . . 68
6.3.5 Key shortcuts . . . 68
6.3.6 Preferences Dialog . . . 69
6.3.7 Displaying Counterexamples . . . 70
6.4 The `replay` Command . . . 72
6.4.1 Obsolete proofs . . . 73
6.4.2 Exit code and options . . . 73
6.4.3 Smoke detector . . . 73
6.5 The `session` Command . . . 74
6.5.1 Command `info` . . . 74
6.5.2 Command `latex` . . . 75
6.5.3 Command `html` . . . 76
6.5.4 Command `update` . . . 78
6.6 The `doc` Command . . . 78
6.6.1 Options . . . 78
6.6.2 Typesetting textual comments . . . 79
6.7 The `pp` Command . . . 79
6.8 The `execute` Command . . . 80
6.8.1 Runtime assertion checking (RAC) . . . 80
6.8.2 Options . . . 80
6.9 The `extract` Command . . . 81
6.10 The `realize` Command . . . 81

6.11 The `wc` Command . . . 81

7 The WhyML Language Reference 83
7.1 Lexical Conventions . . . 83
7.2 Type expressions . . . 85
7.3 Logical expressions . . . 86
7.3.1 Terms and Formulas . . . 86
7.3.2 Specific syntax for collections . . . 88
7.3.3 The "at" and "old" operators . . . 88
7.3.4 Non-standard connectives . . . 89
7.3.5 Conditionals, "let" bindings and pattern-matching . . . 89
7.4 Program expressions . . . 90
7.4.1 Ghost expressions . . . 92
7.4.2 Assignment expressions . . . 92
7.4.3 Auto-dereference: simplified usage of mutable variables . . . 92
7.4.4 Evaluation order . . . 94
7.4.5 Referring to past program states using "at" and "old" operators . . . 94
7.4.6 The "for" loop . . . 94
7.4.7 The "for each" loop . . . 95
7.4.8 Break & Continue . . . 95
7.4.9 Function literals . . . 95
7.4.10 The `any` expression . . . 96
7.5 Modules . . . 96
7.5.1 Record types . . . 98
7.5.2 Algebraic data types . . . 100
7.5.3 Range types . . . 101
7.5.4 Floating-point types . . . 101
7.5.5 Function declarations . . . 102
7.5.6 Module cloning . . . 103
7.6 The Why3 Standard Library . . . 105
7.6.1 Library `int`: mathematical integers . . . 105
7.6.2 Library `array`: array data structure . . . 105

8 The VC Generators 107
8.1 VC generation for program functions . . . 107
8.2 VC generated for type invariants . . . 107
8.3 VC generation and lemma functions . . . 107
8.4 Using strongest post-conditions . . . 108
8.5 Automatic inference of loop invariants . . . 108
8.5.1 Current limitations . . . 109

9 Other Input Formats 111
9.1 micro-C . . . 111
9.1.1 Syntax of micro-C . . . 111
9.1.2 Built-in functions and predicates . . . 113
9.2 micro-Python . . . 114
9.2.1 Syntax of micro-Python . . . 114
9.2.2 Built-in functions and predicates . . . 116
9.2.3 Limitations . . . 116
9.3 MLCFG: function bodies on the style of control-flow graphs . . . 116
9.3.1 Syntax of the MLCFG language . . . 116
9.3.2 An example . . . 118
9.3.3 Error messages . . . 120
9.3.4 Current limitations . . . 120

10 Executing WhyML Programs **121**
10.1 Interpreting WhyML Code . . . 121
10.2 Compiling WhyML to OCaml . . . 121
10.2.1 Examples . . . 122
10.2.2 Extraction of Functors . . . 123
10.2.3 Custom Extraction Drivers . . . 124

11 Interactive Proof Assistants **127**
11.1 Using an Interactive Proof Assistant to Discharge Goals . . . 127
11.2 Theory Realizations . . . 127
11.2.1 Generating a realization . . . 127
11.2.2 Using realizations inside proofs . . . 128
11.2.3 Shipping libraries of realizations . . . 128
11.3 Coq . . . 128
11.4 Isabelle/HOL . . . 129
11.4.1 Installation . . . 129
11.4.2 Usage . . . 129
11.4.3 Using Isabelle server . . . 130
11.4.4 Realizations . . . 130
11.5 PVS . . . 130
11.5.1 Installation . . . 130
11.5.2 Usage . . . 131
11.5.3 Realization . . . 131

12 Technical Informations **133**
12.1 Structure of Session Files . . . 133
12.2 Prover Detection . . . 134
12.3 The `why3.conf` Configuration File . . . 134
12.3.1 Extra Configuration Files . . . 135
12.4 Drivers for External Provers . . . 135
12.5 Adding extra drivers for user theories . . . 135
12.6 Transformations . . . 139
12.7 Proof Strategies . . . 154
12.8 WhyML Attributes . . . 156
12.9 Why3 Metas . . . 157
12.10 Debug Flags . . . 157
12.11 Updating Syntax Error Messages . . . 157

13 Release Notes **159**
13.1 Release Notes for version 1.2: new syntax for “auto-dereference” . . . 159
13.2 Release Notes for version 1.0: syntax changes w.r.t. 0.88 . . . 160
13.3 Release Notes for version 0.80: syntax changes w.r.t. 0.73 . . . 161
13.4 Summary of Changes w.r.t. Why 2 . . . 162

Bibliography **165**

Index **167**

Authors François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, Andrei Paskevich

Version 1.4, February 2022

This work has been partly supported by the U3CAT national ANR project (ANR-08-SEGI-021-08), the Hi-Lite FUI project of the System@tic competitivity cluster, the BWare ANR project (ANR-12-INSE-0010), the Joint Laboratory ProofInUse (ANR-13-LAB3-0007), the CoLiS ANR project (ANR-15-CE25-0001), and the VOCaL ANR project (ANR-15-CE25-008).

CHAPTER

ONE

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 [FM07]. 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.

1.1 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.md` for quick installation instructions, and Section 5 of this document for more detailed instructions.

1.2 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.

1.3 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.

CHAPTER

TWO

GETTING STARTED

2.1 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 `G1`, `G2`, `G3`. 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 Section 3 for detailed explanations. In the following, we show how this file is handled in the Why3 GUI (Section 2.2) then in batch mode using the **why3** executable (Section 2.3).

But before running any Why3 command, you should proceed with the automated detection of external provers (see also Section 5.2). This is done by running the *config* tool on the command line, as follows:

```
why3 config
```

2.2 Getting Started with the GUI

The graphical interface makes it possible 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 6.3 for a more complete description.

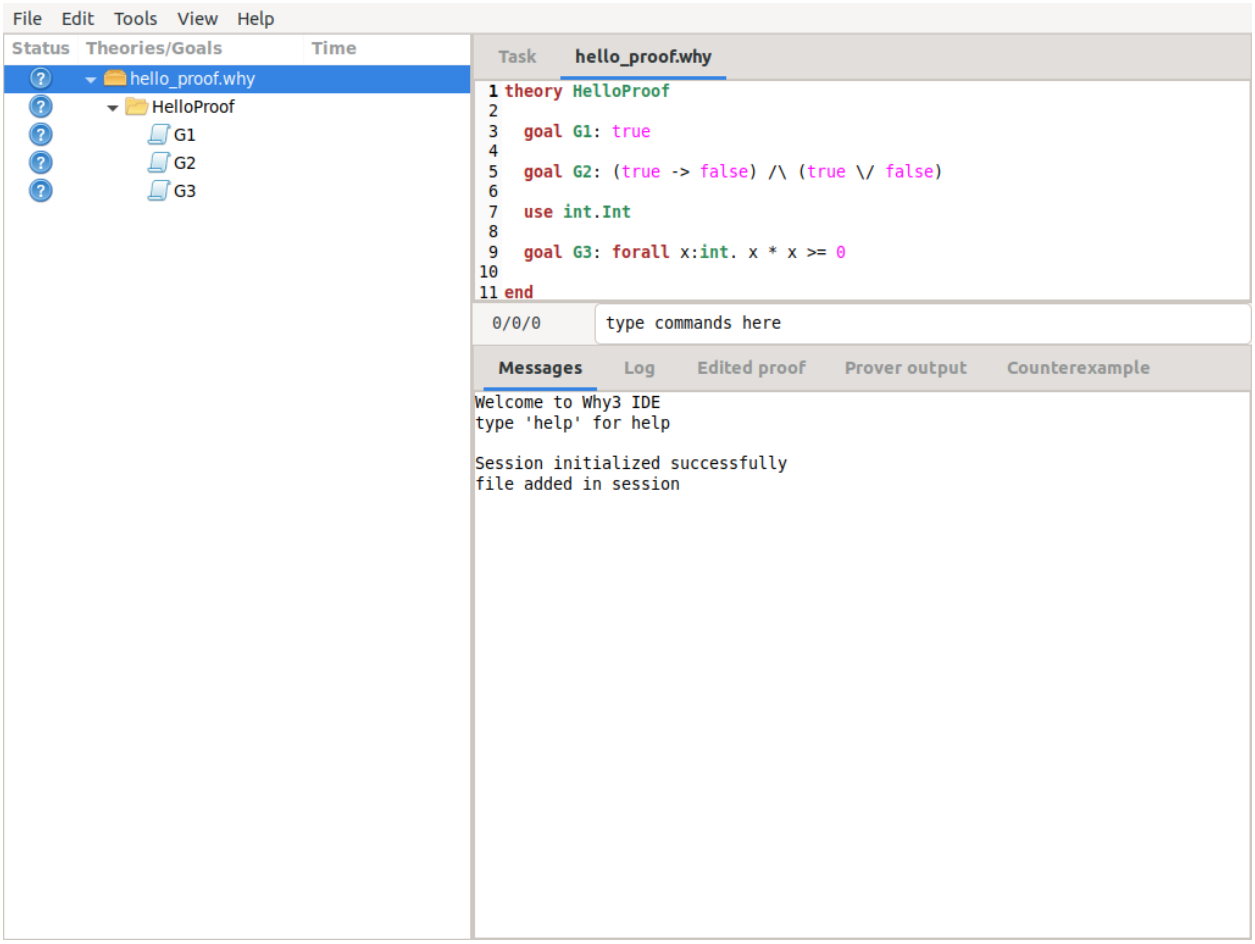

Fig. 2.1: The GUI when started the very first time.

The GUI is launched on the file above as follows:

```
why3 ide hello_proof.why
```

When the GUI is started for the first time, you should get a window that looks like the screenshot of Fig. 2.1. The left part is a tree view that makes it possible to browse inside the files and their theories. The tree view shows that the example is made of a single file containing a single theory containing three goals. The top-right pane displays the content of this file. Now click on the row corresponding to goal `G1`, and then click on the "Task" tab of the top-right pane, so that it displays the corresponding *task*, as show on Fig. 2.2.

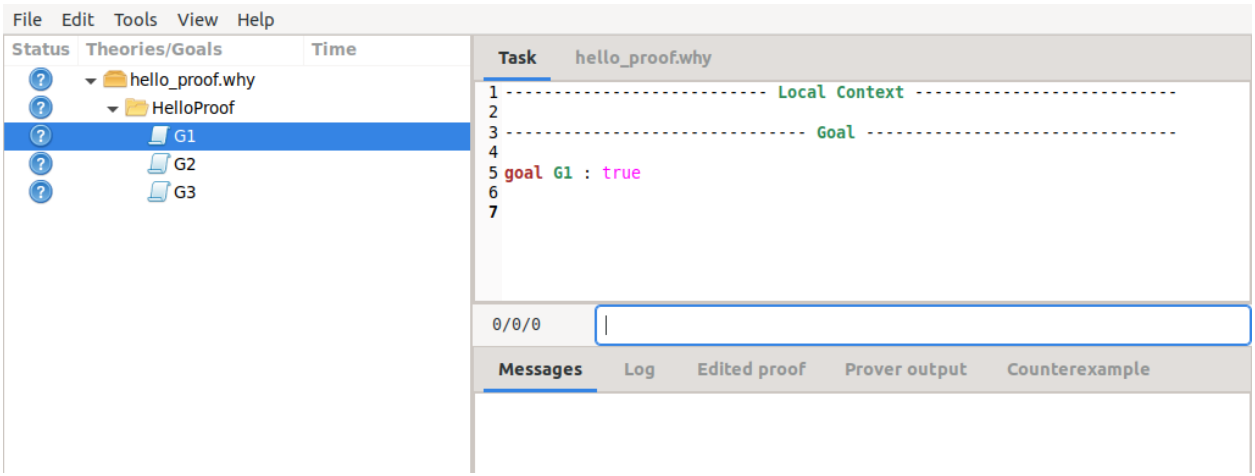

Fig. 2.2: The GUI with goal `G1` selected.

2.2.1 Calling provers on goals

You are now ready to call provers on the goals. (If not done yet, you must perform prover autodetection using *`why3 config`*.) 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 `Control` 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 Fig. 2.3.

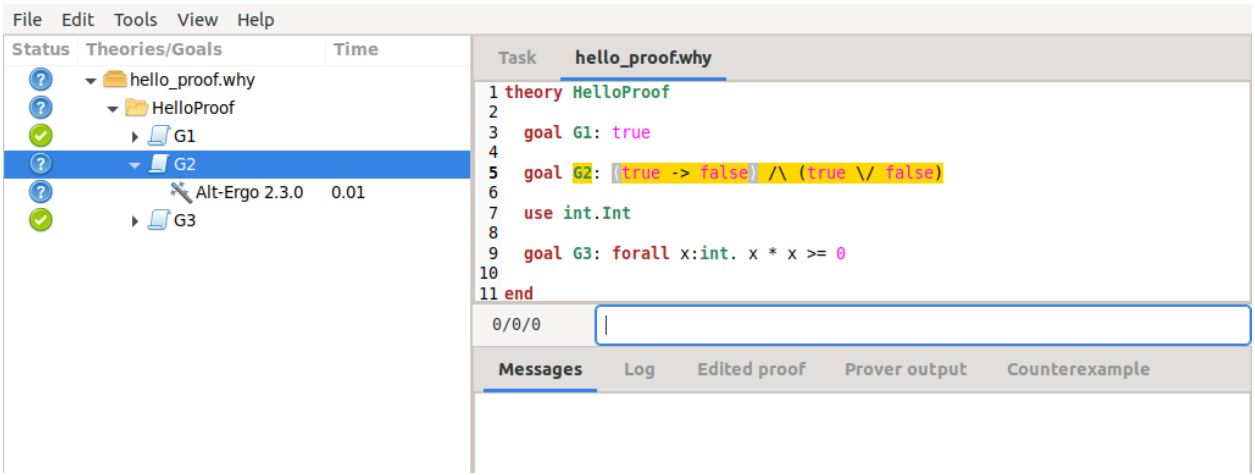

Fig. 2.3: The GUI after running the Alt-Ergo prover on each goal.

Goals `G1` and `G3` 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 `G2` is not proved; it remains marked with a question mark. You could attempt to prove `G2` using another prover, though it is obvious here it will not succeed.

2.2.2 Applying transformations

Instead of calling a prover on a goal, you can apply a transformation to it. Since `G2` 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 `!`. You should now see what is displayed on Fig. 2.4.

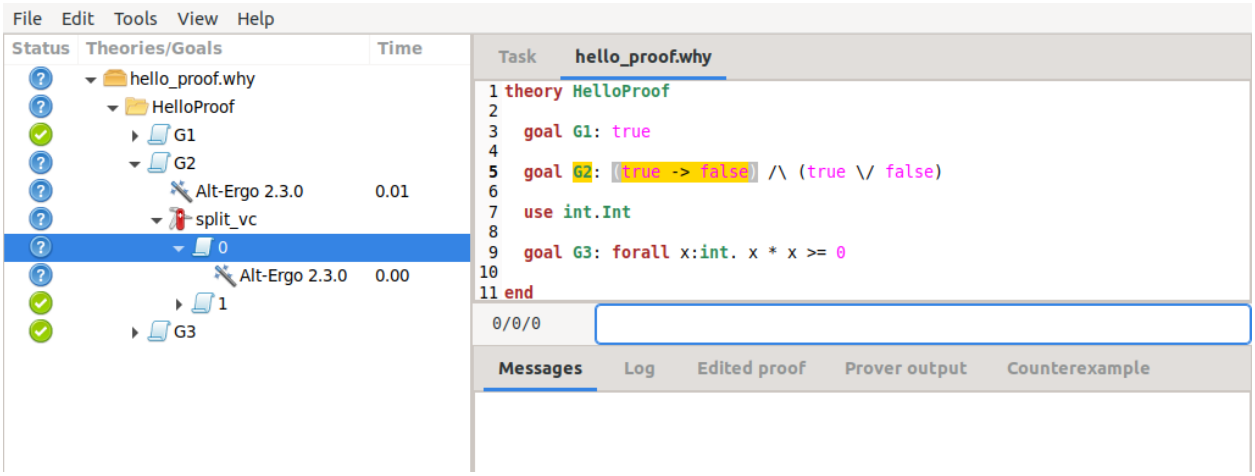

Fig. 2.4: The GUI after splitting goal `G2`.

The first part of goal `G2` is still unproved. As a last resort, you 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 6.3 for details on how to configure this). You get now a regular Coq file to fill in, as shown on Fig. 2.5. 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 11.3 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.

2.2.3 Modifying the input

You can edit the source file, using the corresponding tab in the top-right pane of the GUI. Change the goal `G2` by replacing the first occurrence of `true` by `false`, e.g.,

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

You can refresh the goals using menu *File* → *Save all and Refresh session*, or the shortcut `Control-r`. You get the tree view shown on Fig. 2.6.

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 the goals not yet proved using any prover.

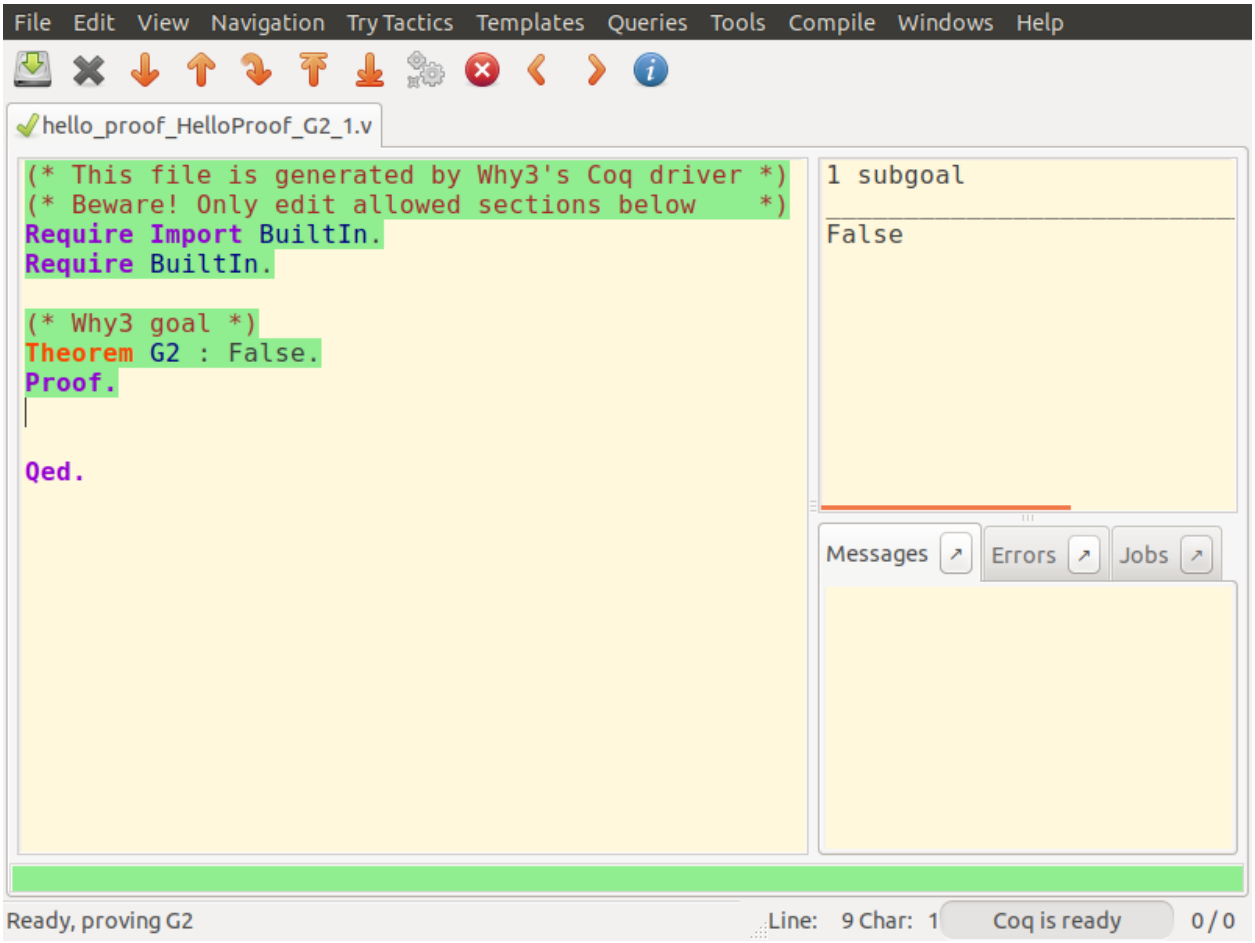

Fig. 2.5: CoqIDE on subgoal 1 of G2.

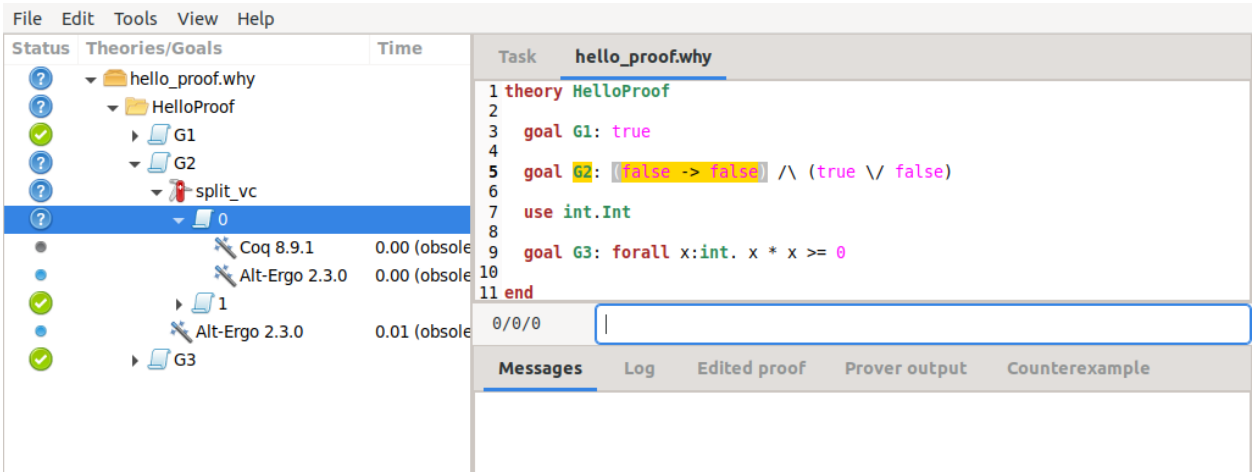

Fig. 2.6: File reloaded after modifying goal G2.

2.2.4 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 valid obsolete proofs*. Notice that replaying can be done in batch mode, using the *why3 replay* command (see Section 6.4) For example, running the replayer on the `hello_proof` example is as follows (assuming `G2` 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 `G2` is not proved.

2.2.5 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 node* or the `Delete` key. Menu *Tools* → *Clean node* 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.

2.3 Getting Started with the Why3 Command

The *why3 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 6.2 for a more complete description of this tool and all its command-line options.

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 [CC08], CVC4 [BCD+11], and Coq [BC04].

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 *why3 prove -P* option is followed by the unique prover identifier (as shown by *why3 --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 G2 and G3.

```
> 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 G2 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 12.6 gives the description of the various transformations available.

CHAPTER

THREE

WHY3 BY EXAMPLES

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 include

- 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 Section 6 for more details regarding command lines.

As an introduction to WhyML, we use a small logical puzzle (Section 3.1) and then the five problems from the VSTTE 2010 verification competition [SM10]. 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`.

3.1 Problem 0: Einstein's Problem

Let us use Why3 to solve a little puzzle known as "Einstein's logic problem". (This Why3 example was contributed by Stéphane Lescuyer.) 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.

```
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
```

(continues on next page)

(continued from previous page)

```
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 (see Section 7.5.6). 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 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 Einstein
  use 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`.

3.2 Problem 1: Sum and Maximum

Let us now move to the problems of the VSTTE 2010 verification competition [SM10]. 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 int.Int
```

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

```
use ref.Ref
use 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`:

```
!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`, shown below.

```
module MaxAndSum

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

```

(continues on next page)

(continued from previous page)

```
  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
```

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.

Note: It is of course possible to *execute* the code to test it, before or after you prove it correct. This is detailed in Section 10.1.

Auto-dereference

Why3 features an auto-dereferencing mechanism, which simplifies the use of references. When a reference is introduced using `let ref x` (instead of `let x = ref`), the use of operator `!` to access its value is not needed anymore. For instance, we can rewrite the program above as follows (the contract being unchanged and omitted):

```
let max_sum (a: array int) (n: int) : (sum: int, max: int)
= let ref sum = 0 in
  let ref max = 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
```

Note that use of operator `<-` for assignment (instead of `:=`) and the absence of `!` both in the loop invariant and in the program. See Section 13.1 for more details about the auto-dereferencing mechanism.

3.3 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 $N - 1$, 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 int.Int
  use 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 below. The verification conditions for function `inverting` are easily discharged automatically, thanks to the lemma.

```
module InvertingAnInjection

  use int.Int
  use 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) =
```

(continues on next page)

(continued from previous page)

```
    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
```

3.4 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.

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 nth element of a list. The latter returns an option type, depending on whether the index is meaningful or not.

```
module SearchingALinkedList
  use int.Int
  use 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 giving 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 below. The verification conditions are all discharged automatically.

```
module SearchingALinkedList

  use 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
```

(continues on next page)

(continued from previous page)

```
    ensures { (0 <= result < length l /\ zero_at l result)
           \/ (result = length l /\ no_zero l) }
  = search 0 l

end
```

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 ref.Ref
use 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
```

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.

3.5 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 int.Int
  use 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) };
```

(continues on next page)

(continued from previous page)

```
    False
  end
```

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

```
module NQueens
  use int.Int
  use 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
```

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 }
```

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 below. 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.

```
  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
```

3.6 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* [Oka98] 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 int.Int
  use option.Option
  use export list.ListRich
```

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 satisfies the invariant at any function entry and it requires that any queue satisfies the invariant at function exit (be the queue created or modified). The `by` clause ensures the non-vacuity of this type with invariant. If you omit it, a goal with an existential statement is generated. (See Section 7.5.1 for more details about record types with invariants.)

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 below. The verification conditions are all discharged automatically.

```
module AmortizedQueue

  use int.Int
  use option.Option
  use 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 }
```

(continues on next page)

(continued from previous page)

```
  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
```

CHAPTER

FOUR

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-1.4/.

We assume the reader has a fair knowledge of the OCaml language. Notice that the Why3 library must be installed, see Section 5.1.3. The OCaml code given below is available in the source distribution in directory `examples/use_api/` together with a few other examples.

4.1 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 \/ 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 /\ B -> 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.

4.2 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 \/ 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")
```

(continues on next page)

(continued from previous page)

```
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
```

4.3 Calling External Provers

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

```
(* reads the default config file *)
let config : Whyconf.config = Whyconf.init_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
```

(continues on next page)

(continued from previous page)

```
  if Whyconf.Mprover.is_empty provers then begin
      eprintf "Prover Alt-Ergo not installed or not configured@.";
      exit 1
    end else begin
      printf "Versions of Alt-Ergo found:";
      Whyconf.(Mprover.iter (fun k _ -> printf " %s" k.prover_version) provers);
      printf "@.";
      (* returning an arbitrary one *)
      snd (Whyconf.Mprover.max_binding provers)
    end
```

We could also get a specific version with

```
(* Specific version 2.3.0 of Alt-Ergo in the config file *)
let _ : Whyconf.config_prover =
  let fp = Whyconf.parse_filter_prover "Alt-Ergo,2.3.0" in
  let provers = Whyconf.filter_provers config fp in
  if Whyconf.Mprover.is_empty provers then begin
      eprintf "Prover Alt-Ergo 2.3.0 not installed or not configured, using version %s␣
↪instead@."
        Whyconf.(alt_ergo.prover.prover_version) ;
    alt_ergo (* we don't want to fail this time *)
  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
  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 ?json:None) 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_models : (prover_answer * model) list;
}
```

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 cannot determine if the task is valid; the string parameter *msg* indicates some extra information.
- `Failure` *msg*: the prover reports a failure, 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 (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
```

4.4 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
```

4.5 Building Quantified Formulas

To illustrate how to build quantified formulas, let us consider the formula $\forall x : int.x \cdot x \geq 0$. 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 $x \cdot x \geq 0$ 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
```

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

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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
  (* 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.

4.7 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 =
```

(continues on next page)

(continued from previous page)

```
  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.

4.8 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.

4.9 ML Programs

One can build WhyML programs starting at different steps of the WhyML pipeline (parsing, typing, VC generation). We present here two choices. The first is to build an untyped syntax trees, and then call the Why3 typing procedure to build typed declarations. The second choice is to directly build the typed declaration. The first choice use concepts similar to the WhyML language but errors in the generation are harder to debug since they are lost inside the typing phase, the second choice use more internal notions but it is easier to pinpoint the functions wrongly used. Section 4.9.1 and Section 4.9.4 follow choice one and Section 4.9.5 choice two.

4.9.1 Untyped syntax tree

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 4.3, open the OCaml module `Ptree` ("parse tree") which contains the type constructors for the parsing trees, and finally the OCaml module `Ptree_helpers` which contains helpers for building those trees and a more concise and friendly manner than the low-level constructors. The latter two OCaml modules are documented in the online API documentation, respectively for Ptree and Ptree_helpers.

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

Each of our example programs will build a module. Let us consider the Why3 code.

```
module M1
  use int.Int
  goal g : 2 + 2 = 4
end
```

The Ocaml code that programmatically builds it is as follows.

```
let mod_M1 =
  (* use int.Int *)
  let use_int_Int = use ~import:false (["int";"Int"]) in
  (* goal g : 2 + 2 = 4 *)
  let g =
    let two = tconst 2 in
    let four = tconst 4 in
    let add_int = qualid ["Int";Ident.op_infix "+"] in
    let two_plus_two = tapp add_int [two ; two] in
    let eq_int = qualid ["Int";Ident.op_infix "="] in
    let goal_term = tapp eq_int  [four ; two_plus_two] in
    Dprop(Decl.Pgoal, ident "g", goal_term)
  in
  (ident "M1",[use_int_Int ; g])
```

Most of the code is not using directly the `Ptree` constructors but instead makes uses of the helper functions that are given in the `Ptree_helpers` module. Notice `ident` which builds an identifier (type `Ptree.ident`) optionally with attributes and location and `use` which lets us import some other modules and in particular the ones from the standard library. At the end, our module is no more than the identifier and a list of two declarations (`Ptree.decl list`).

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

```
module M2
  let f (x:int) : int
    requires { x=6 }
    ensures { result=42 }
   = x*7
end
```

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

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

let mod_M2 =
  (* use int.Int *)
  let use_int_Int = use ~import:false (["int";"Int"]) in
  (* f *)
  let f =
    let id_x = ident "x" in
    let pre = tapp eq_symb [tvar (Qident id_x); tconst 6] in
    let result = ident "result" in
    let post = tapp eq_symb [tvar (Qident result); tconst 42] in
    let spec = {
      sp_pre = [pre];
```

(continues on next page)

(continued from previous page)

```
      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 = eapp mul_int [evar (Qident id_x); econst 7] in
    let f =
      Efun(one_binder ~pty:int_type "x", None, pat Pwild,
           Ity.MaskVisible, spec, body)
    in
    Dlet(ident "f",false,Expr.RKnone, expr f)
  in
  (ident "M2",[use_int_Int ; f])
```

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

```
module M3
  let f() : int
     requires { true }
     ensures { result >= 0 }
   = let x = ref 42 in !x
end
```

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 = qualid ["Int";Ident.op_infix ">="]

let mod_M3 =
  (* use int.Int *)
  let use_int_Int = use ~import:false (["int";"Int"]) in
  (* use ref.Ref *)
  let use_ref_Ref = use ~import:false (["ref";"Ref"]) in
  (* f *)
  let f =
    let result = ident "result" in
    let post = term(Tidapp(ge_int,[tvar (Qident result);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;
```

(continues on next page)

(continued from previous page)

```
      sp_partial = false;
    }
    in
    let body =
      let e1 = eapply (evar (qualid ["Ref";"ref"])) (econst 42) in
      let id_x = ident "x" in
      let qid = qualid ["Ref";Ident.op_prefix "!"] in
      let e2 = eapply (evar qid) (evar (Qident id_x)) in
      expr(Elet(id_x,false,Expr.RKnone,e1,e2))
    in
    let f = Efun(unit_binder (),None,pat Pwild,Ity.MaskVisible,spec,body)
    in
    Dlet(ident "f",false,Expr.RKnone, expr f)
  in
  (ident "M3",[use_int_Int ; use_ref_Ref ; f])
```

The next example makes use of arrays.

```
module M4
  let f (a:array int) : unit
    requires { a.length >= 1 }
    ensures { a[0] = 42 }
   = a[0] <- 42
end
```

The corresponding OCaml code is as follows.

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

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

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

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

let mod_M4 =
  (* use int.Int *)
  let use_int_Int = use ~import:false (["int";"Int"]) in
  (* use array.Array *)
  let use_array_Array = use ~import:false (["array";"Array"]) in
  (* use f *)
  let f =
    let id_a = ident "a" in
    let pre =
      tapp ge_int [tapp length [tvar (Qident id_a)]; tconst 1]
    in
    let post =
      tapp eq_symb [tapp array_get [tvar (Qident id_a); tconst 0];
                    tconst 42]
    in
    let spec = {
      sp_pre = [pre];
```

(continues on next page)

(continued from previous page)

```
      sp_post = [Loc.dummy_position,[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 =
      eapp array_set [evar (Qident id_a); econst 0; econst 42]
    in
    let f = Efun(one_binder ~pty:array_int_type "a",
                 None,pat Pwild,Ity.MaskVisible,spec,body)
    in
    Dlet(ident "f", false, Expr.RKnone, expr f)
  in
  (ident "M4",[use_int_Int ; use_array_Array ; f])
```

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.

```
let mlw_file = Modules [mod_M1 ; mod_M2 ; mod_M3 ; mod_M4]
```

4.9.2 Alternative, top-down, construction of parsing trees

The way we build our modules above is somehow bottom-up: builds the terms and the program expressions, then the declarations that contain them, then the modules containing the latter declarations. An alternative provided by other helpers is to build those modules in a top-down way, which may be more natural since this the order they occur in the concrete syntax. We show below how to construct a similar list of module as above, with only the last module for conciseness:

```
let mlw_file =
  let uc = F.create () in
  let uc = F.begin_module uc "M5" in
  let uc = F.use uc ~import:false ["int";"Int"] in
  let uc = F.use uc ~import:false ["array";"Array"] in
  let uc = F.begin_let uc "f" (one_binder ~pty:array_int_type "a") in
  let id_a = Qident (ident "a") in
  let pre = tapp ge_int [tapp length [tvar id_a]; tconst 1] in
  let uc = F.add_pre uc pre in
  let post =
    tapp eq_symb [tapp array_get [tvar id_a; tconst 0];
                  tconst 42]
  in
  let uc = F.add_post uc post in
  let body = eapp array_set [evar id_a; econst 0; econst 42] in
  let uc = F.add_body uc body in
  let uc = F.end_module uc in
```

(continues on next page)

(continued from previous page)

```
  F.get_mlw_file uc
```

The construction above is functional, in the sense that the `uc` variable holds the necessary data for the modules under construction. For simplicity it is also possible to use an imperative variant which transparently handles the state of modules under construction.

```
let mlw_file =
  I.begin_module "M6";
  I.use ~import:false ["int";"Int"];
  I.use ~import:false ["array";"Array"];
  I.begin_let "f" (one_binder ~pty:array_int_type "a");
  let id_a = Qident (ident "a") in
  let pre = tapp ge_int [tapp length [tvar id_a]; tconst 1] in
  I.add_pre pre;
  let post =
    tapp eq_symb [tapp array_get [tvar id_a; tconst 0];
                  tconst 42]
  in
  I.add_post post;
  let body = eapp array_set [evar id_a; econst 0; econst 42] in
  I.add_body body;
  I.end_module ();
  I.get_mlw_file ()
```

Beware though that the latter approach is not thread-safe and cannot be used in re-entrant manner.

4.9.3 Using the parsing trees

Module `Mlw_printer` provides functions to print elements of `Ptree` in concrete whyml syntax.

```
let () = printf "%a@." Mlw_printer.pp_mlw_file mlw_file
```

The typing of the modules is carried out by function `Typing.type_mlw_file`, which produces a mapping of module names to typed modules.

```
let mods = Typing.type_mlw_file env [] "myfile.mlw" mlw_file
```

Typing errors are reported by exceptions `Located of position * exn` from module `Loc`. However, the positions in our declarations, which are provided by the exception, cannot be used to identify the position in the (printed) program, because the locations do not correspond to any concrete syntax.

Alternatively, we can give every `Ptree` element in our declarations above a unique location (for example using the function `Mlw_printer.next_pos`). When a located error is encountered, the function `Mlw_printer.with_marker` can then be used to instruct `Mlw_printer` to insert the error as a comment just before the syntactic element with the given location.

```
let _mods =
  try
    Typing.type_mlw_file env [] "myfile.mlw" mlw_file
  with Loc.Located (loc, e) -> (* A located exception [e] *)
    let msg = asprintf "%a" Exn_printer.exn_printer e in
    printf "%a@."
```

(continues on next page)

(continued from previous page)

```
      (Mlw_printer.with_marker ~msg loc Mlw_printer.pp_mlw_file)
      mlw_file;
    exit 1
```

Finally, we can then construct the proofs tasks for our typed 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



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,2.3.0" in
  let provers = Whyconf.filter_provers config fp in
  if Whyconf.Mprover.is_empty provers then begin
    eprintf "Prover Alt-Ergo 2.3.0 not installed or not configured@.";
    exit 1
  end else
    snd (Whyconf.Mprover.max_binding provers)

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

let () =
  List.iteri (fun i t ->
      let call = Driver.prove_task ~limit:Call_provers.empty_limit
          ~command:alt_ergo.Whyconf.command alt_ergo_driver t in
      let r = Call_provers.wait_on_call call in
      printf "@[On task %d, alt-ergo answers %a@." (succ i)
        (Call_provers.print_prover_result ?json:None) r)
    my_tasks
```

4.9.4 Use attributes to infer loop invariants

In this section we build a module containing a let declaration with a while loop and an attribute that triggers the inference of loop invariants during VC generation. For more information about the inference of loop invariants refer to Section 5.5 and Section 8.5. The examples shown below are available in the file `examples/use_api/mlw_tree_infer_invs.ml`.

We build an environment and define the some helper functions exactly as in Section 4.9.1. Additionally we create two other helper functions as follows:

```
(* ... *)
let pat_wild = mk_pat Pwild
let pat_var id = mk_pat (Pvar id)

let mk_ewhile e1 i v e2 = mk_expr (Ewhile (e1,i,v,e2))
```

Our goal is now to build a program equivalent to the following. Note that the let declaration contains an attribute *[@infer]* which will trigger the inference of loop invariants during VC generation (make sure that the why3 library was compiled with support for *infer-loop*, see Section 5.5 for more information).

```
module M1
  use int.Int
  use ref.Refint
  let f [@infer] (x:ref int) : unit
    requires { !x <= 100 }
    ensures  { !x = 100 }
  = while (!x < 100) do
      variant { 100 - !x }
      incr x
    done
end
```

The OCaml code that builds such a module is shown below.

```
let int_type_id  = mk_qualid ["int"]
let int_type     = PTtyapp(int_type_id,[])
let ref_int_type = PTtyapp (mk_qualid ["ref"], [int_type])
let ref_access   = mk_qualid ["Refint"; Ident.op_prefix "!"]
let ref_assign   = mk_qualid ["Refint"; Ident.op_infix ":="]
let ref_int_incr = mk_qualid ["Refint"; "incr"]
let l_int        = mk_qualid ["Int";Ident.op_infix "<"]
let le_int       = mk_qualid ["Int";Ident.op_infix "<="]
let plus_int     = mk_qualid ["Int";Ident.op_infix "+"]
let minus_int    = mk_qualid ["Int";Ident.op_infix "-"]
let eq_symb      = mk_qualid [Ident.op_infix "="]

let mod_M1 =
  (* use int.Int *)
  let use_int_Int    = use_import (["int";"Int"]) in
  let use_ref_Refint = use_import (["ref";"Refint"]) in
  (* f *)
  let f =
    let id_x  = mk_ident "x" in
    let var_x = mk_var id_x in
```

(continues on next page)

(continued from previous page)

```
    let t_x   = mk_tapp ref_access [var_x] in
    let pre   = mk_tapp le_int [t_x; mk_tconst 100] in
    let post  = mk_tapp eq_symb [t_x; mk_tconst 100] in
    let spec  = {
      sp_pre     = [pre];
      sp_post    = [Loc.dummy_position,[pat_var (mk_ident "result"), post]];
      sp_xpost   = [];
      sp_reads   = [];
      sp_writes  = [];
      sp_alias   = [];
      sp_variant = [];
      sp_checkrw = false;
      sp_diverge = false;
      sp_partial = false;
    }
    in
    let var_x      = mk_evar (Qident id_x) in
    (* !x *)
    let e_x        = mk_eapp ref_access [var_x] in
    (* !x < 100 *)
    let while_cond = mk_eapp l_int [e_x; mk_econst 100] in
    (* 100 - !x *)
    let while_vari = mk_tapp minus_int [mk_tconst 100; t_x], None in
    (* incr x *)
    let incr       = mk_apply (mk_evar ref_int_incr) var_x in
    (* while (!x < 100) do variant { 100 - !x } incr x done *)
    let while_loop = mk_ewhile while_cond [] [while_vari] incr in
    let f =
      Efun(param1 id_x ref_int_type, None, mk_pat Pwild,
           Ity.MaskVisible, spec, while_loop)
    in
    let attr = ATstr (Ident.create_attribute "infer") in
    let id = { (mk_ident "f") with id_ats = [attr] } in
    Dlet(id,false,Expr.RKnone, mk_expr f)
  in
  (mk_ident "M1",[use_int_Int; use_ref_Refint; f])
```

Optionally, the debugging flags mentioned in Section 8.5 can be enabled by using the API as follows (the line(s) corresponding to the desired flag(s) should be uncommented).

```
(* let () = Debug.set_flag Infer_cfg.infer_print_ai_result *)
(* let () = Debug.set_flag Infer_cfg.infer_print_cfg *)
(* let () = Debug.set_flag Infer_loop.print_inferred_invs *)
```

Another option is to register a function to be executed immediately after the invariants are inferred. The function should have type `(expr * term) list -> unit`, where `expr` corresponds to a while loop and `term` to the respective inferred invariant. The function can be registered using the function `Infer_loop.register_hook`.

In the following example a sequence of three functions are registered. The first function will write the invariants to the standard output, the second to a file named *inferred_invs.out*, and the third will save the inferred invariants in `inv_terms`.

```
let print_to_std invs =
```

(continues on next page)

(continued from previous page)

```
  let print_inv fmt (_,t) = Pretty.print_term fmt t in
  Format.printf "The following invariants were generated:@\n%a@."
    (Pp.print_list Pp.newline2 print_inv) invs

let print_to_file invs =
  let print_inv fmt (_,t) = Pretty.print_term fmt t in
  let fmt = Format.formatter_of_out_channel (open_out "inferred_invs.out") in
  Format.fprintf fmt "Generated invariants:@\n%a@."
    (Pp.print_list Pp.newline2 print_inv) invs

let inv_terms = ref []

let save_invs invs =
  inv_terms := List.map snd invs

let () =
  Infer_loop.register_hook (fun i ->
      print_to_std i; print_to_file i; save_invs i)
```

Finally the code for closing the modules, printing it to the standard output, typing it, and so on is exactly the same as in the previous section, thus we omit it in here. Note that in practice, the invariants are only inferred when invoking `Typing.type_mlw_file`.

4.9.5 Typed declaration

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

The first step to build an environment as already illustrated in Section 4.3.

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

To write our programs, we need to import some other modules from the standard library integers and references. The only subtleties is to get logic functions from the logical part of the modules `mod_theory.Theory.th_export` and the program functions from `mod_export`.

```
let int_module : Pmodule.pmodule =
  Pmodule.read_module env ["int"] "Int"

let ge_int : Term.lsymbol =
  Theory.ns_find_ls int_module.Pmodule.mod_theory.Theory.th_export
    [Ident.op_infix ">="]

let ref_module : Pmodule.pmodule =
  Pmodule.read_module env ["ref"] "Ref"

let ref_type : Ity.itysymbol =
  Pmodule.ns_find_its ref_module.Pmodule.mod_export ["ref"]

(* the "ref" function *)
```

(continues on next page)

(continued from previous page)

```
let ref_fun : Expr.rsymbol =
  Pmodule.ns_find_rs ref_module.Pmodule.mod_export ["ref"]

(* the "!" function *)
let get_fun : Expr.rsymbol =
  Pmodule.ns_find_rs ref_module.Pmodule.mod_export [Ident.op_prefix "!"]
```

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
```

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

```
let d2 =
  let id = Ident.id_fresh "f" in
  let post =
    let result =
      Term.create_vsymbol (Ident.id_fresh "result") Ty.ty_int
    in
    let post = Term.ps_app ge_int [Term.t_var result; Term.t_nat_const 0] in
    Ity.create_post result post
  in
  let body =
    (* building expression "ref 42" *)
    let e =
      let c0 = Expr.e_const (Constant.int_const_of_int 42) Ity.ity_int in
      let refzero_type = Ity.ity_app ref_type [Ity.ity_int] [] in
      Expr.e_app ref_fun [c0] [] refzero_type
    in
    (* building the first part of the let x = ref 42 *)
    let id_x = Ident.id_fresh "x" in
    let letdef, var_x = Expr.let_var id_x e in
    (* building expression "!x" *)
    let bang_x = Expr.e_app get_fun [Expr.e_var var_x] [] Ity.ity_int in
    (* the complete body *)
    Expr.e_let letdef bang_x
  in
  let arg_unit =
    let unit = Ident.id_fresh "unit" in
    Ity.create_pvsymbol unit Ity.ity_unit
  in
  let def,_rs = Expr.let_sym id
      (Expr.c_fun [arg_unit] [] [post] Ity.Mxs.empty Ity.Mpv.empty body) in
  Pdecl.create_let_decl def
```

Having declared all the programs we wanted to write, we can now create the module and generate the VCs.

```
let mod2 =
  let uc : Pmodule.pmodule_uc =
    Pmodule.create_module env (Ident.id_fresh "MyModule")
```

(continues on next page)

(continued from previous page)

```
  in
  let uc = Pmodule.use_export uc int_module in
  let uc = Pmodule.use_export uc ref_module in
  let uc = Pmodule.add_pdecl ~vc:true uc d2 in
  Pmodule.close_module uc
```

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 =
  Task.split_theory mod2.Pmodule.mod_theory None None

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 1
  end else
    snd (Whyconf.Mprover.max_binding provers)

let alt_ergo_driver : Driver.driver =
  try
    Whyconf.load_driver main env alt_ergo
  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)
```

(continues on next page)

(continued from previous page)

```
      in
      printf "@[On task %d, alt-ergo answers %a@."
             i (Call_provers.print_prover_result ?json:None) r;
      i+1
     )
     1 my_tasks
  in ()
```

4.10 Generating counterexamples

That feature is presented in details in Section 6.3.7, which 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 4.1 to display counterexamples.

4.10.1 Attributes and locations on identifiers

For variables to be used for counterexamples they need to contain an attribute called *model_trace* and a location. This attribute 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 will not be displayed. For example, an assignment of the source language such as the following will probably trigger the creation of an identifier (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 identifier with a location and a `model_trace`.

4.10.2 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 []
```

(continues on next page)

(continued from previous page)

```
(* 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 occurring 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.

4.10.3 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, CVC4,1.7 answers %a@."
    (Call_provers.print_prover_result ?json:None) result1

let () = printf "Model is %t@."
    (fun fmt ->
       match result1.Call_provers.pr_models with
       | [(_,m)] -> (* TODO select_model *)
           Model_parser.print_model_json ?me_name_trans:None ?vc_line_trans:None fmt m
       | _ -> fprintf fmt "unavailable")
```

CHAPTER

FIVE

COMPILATION, INSTALLATION

5.1 Installing Why3

5.1.1 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 5.2 to install external provers.

5.1.2 Installation via Docker

Instead of compiling Why3, one can also execute a precompiled version (for *amd64* architecture) using Docker. The image containing Why3 as well as a few provers can be recovered using

```
docker pull registry.gitlab.inria.fr/why3/why3:1.4.1
docker tag  registry.gitlab.inria.fr/why3/why3:1.4.1 why3
```

Let us suppose that there is a file `foo.mlw` in your current directory. If you want to verify it using Z3, you can type

```
docker run --rm --volume `pwd`:/data --workdir /data why3 prove foo.mlw -P z3
```

If you want to verify `foo.mlw` using the graphical user interface, the invocation is slightly more complicated as the containerized application needs to access your X server:

```
docker run --rm --network host --user `id -u` --volume $HOME/.Xauthority:/home/guest/.
↪Xauthority --env DISPLAY=$DISPLAY --volume `pwd`:/data --workdir /data why3 ide foo.mlw
```

It certainly makes sense to turn this command line into a shell script for easier use:

```
#!/bin/sh
exec docker run --rm --network host --user `id -u` --volume $HOME/.Xauthority:/home/
↪guest/.Xauthority --env DISPLAY=$DISPLAY --volume `pwd`:/data --workdir /data why3 "$@"
```

It is also possible to run the graphical user interface from within a web browser, thus alleviating the need for a X server. To do so, just set the environment variable `WHY3IDE` to `web` and publish port 8080:

```
docker run --rm -p 8080:8080 --env WHY3IDE=web --user `id -u` --volume `pwd`:/data --
↪workdir /data why3 ide foo.mlw
```

You can now point your web browser to http://localhost:8080/. As before, this can be turned into a shell script for easier use:

```
#!/bin/sh
exec docker --rm -p 8080:8080 --env WHY3IDE=web --user `id -u` --volume `pwd`:/data --
↪workdir /data why3 "$@"
```

5.1.3 Installation from Source Distribution

In short, installation from sources proceeds as follows.

```
./configure
make
make install
```

After unpacking the distribution, go to the newly created directory `why3-1.4`. 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 11.2), 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 11.5) or Isabelle (Section 11.4) 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 5.2 below)
2. run *why3 config*

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

Installing Why3 is not mandatory. It can be configured in a way such that it can be used from its compilation directory:

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

The Why3 executable files are then available in the subdirectory `bin/`. This directory can be added to 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
```

Beware that if your OCaml installation relies on Opam installed in your own user space, then `make install-lib` should *not* be run as super-user.

Removing Installation

Removing installation can be done using

```
make uninstall
make uninstall-lib
```

5.2 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
```

It scans your `PATH` for provers and updates your configuration file (see Section 6.1) accordingly.

5.2.1 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, the command *why3 config add-prover* can be used to specify names of prover executables:

```
why3 config add-prover CVC4 /usr/local/bin/cvc4-dev cvc4-dev
```

the first argument (here CVC4) must be one of the known provers. The list of these names can be obtained using *why3 config list-supported-provers*. They can also be found in the file `provers-detection-data.conf`, typically located in `/usr/local/share/why3` after installation. See Section 12.2 for details.

5.2.2 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 *Files* → *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 6.5.

5.3 Configure Editors for editing WhyML sources

The Why3 distributions come with some configuration files for Emacs and for Vim. These files are typically installed in the shared data directory, which is given by

```
why3 --print-datadir
```

5.3.1 Emacs

The Why3 distributions come with a mode for Emacs in a file `why3.el`. That file is typically found in sub-directory `emacs`. Under OPAM, this file is installed in a shared directory `emacs/site-lisp` for all OPAM packages. Here is a sample Emacs-Lisp code that can be added to your `.emacs` configuration file.

```
(setq why3-share (if (boundp 'why3-share) why3-share (ignore-errors (car␣
↪(process-lines "why3" "--print-datadir")))))
(setq why3el
 (let ((f (expand-file-name "emacs/why3.elc" why3-share)))
   (if (file-readable-p f) f
     (let ((f (expand-file-name "emacs/site-lisp/why3.elc" opam-share)))
       (if (file-readable-p f) f nil)))))
(when why3el
  (require 'why3)
  (autoload 'why3-mode why3el "Major mode for Why3." t)
  (setq auto-mode-alist (cons '("\\.mlw$" . why3-mode) auto-mode-alist)))
```

5.3.2 Vim

Some configuration files are present in the share data directory, under sub-directory `vim`.

5.4 Configure Shells for auto-completion of Why3 command arguments

Some configuration files for shells are distributed in the shared data directory, which is given by `why3 --print-data-dir`.

There are configuration files for `bash` and `zsh`.

The configuration for `bash` can be made from Why3 sources using

```
sudo make install-bash
```

or directly doing

```
sudo /usr/bin/install -c `why3 --print-datadir`/bash/why3 /etc/bash_completion.
↪d
```

5.5 Inference of Loop Invariants

This section shows how to install *infer-loop*, an utility based on *abstract interpretation* to infer loop invariants [Bau17]. This is still work in progress and many features are still very limited.

The `infer-loop` utility has the following OCaml dependencies.

- `apron`: can be installed using `opam`.
- `camllib`: can be installed using `opam`.
- `fixpoint`: follow instructions below.

The `apron` and `camllib` libraries can be installed using `opam`. The `fixpoint` library is not available in `opam`, but it can be easily compiled and installed using the source code. The following commands are just an example of how the library can be compiled and installed, and can be performed in any directory.

```
svn co svn://scm.gforge.inria.fr/svnroot/bjeannet/pkg/fixpoint
cd fixpoint/trunk/
cp Makefile.config.model Makefile.config
# if required make modifications to Makefile.config
make all     # compiles
make install # uses ocamlfind to install the library
```

By default the *infer-loop* mechanism is not compiled and integrated with Why3. So, once the dependencies above are installed, the configuration script of Why3 should enable the compilation of the `infer-loop` utility. This can be done by passing to the Why3 configure script the `--enable-infer` flag, as follows:

```
./configure --enable-infer
# ...
# Components
# ...
#    Invariant inference(exp): yes
# ...
```

The line `Invariant inference(exp)` indicates whether the dependencies are correctly installed and whether the flag mentioned above was selected. After the compilation, the loop inference mechanism should be available. See Section 8.5 for more details.

CHAPTER

SIX

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 follows:

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

The following commands are available:

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

doc Render a WhyML file as HTML.

execute Perform a symbolic execution of a WhyML file.

extract Generate an OCaml program corresponding to a WhyML file.

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

pp Pretty-print WhyML definitions (formatting `.mlw` files or printing inductive definitions to LaTeX).

prove Read a WhyML input file and call provers, on the command-line.

realize Generate the skeleton of an interactive proof for a WhyML file.

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

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

wc Give some token statistics about a WhyML file.

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>, **--library**=<dir>
: Add <dir> in the load path, to search for theories.

-C <file>, **--config**=<file>
: Read the configuration from the given file. See Section 12.3.

--extra-config=<file>
: Read additional configuration from the given file. See Section 12.3.

--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. See also Section 12.9 for a description of some of those metas.

--debug-all
Set all debug flags (except flags that change the behavior).

--debug=<flag>,...
Set some specific debug flags. See also Section 12.10 for a description of some of those flags.

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

WHY3CONFIG
Indicate where to find the `why3.conf` file. Can be overwritten using the *--config* option.

6.1 The `config` Command

Why3 must be configured to access external provers. Typically, this is done by running *why3 config detect*. This command must be run every time a new prover is installed.

The provers known by Why3 are described in the 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 prover detection is stored in the user's configuration file (see Section 12.3). Only the version of the provers is stored; the actual configuration of the provers, shortcuts, strategies, and editors, are regenerated at each startup of a Why3. This configuration can be inspected with the command *why3 config show*.

If a supported prover is not automatically recognized by *why3 config detect*, the command *why3 config add-prover* can be used to add it.

The available subcommands are as follows:

config add-prover Manually register a prover.

config detect Automatically detect installed provers.

config list-supported-provers List the names of all supported provers.

config show Show the expanded version of the configuration file.

Only the first two commands modify the configuration file.

6.1.1 Command `add-prover`

This commands adds a prover to the configuration. It is invoked as follows.

```
why3 config add-prover <name> <file> [<shortcut>]
```

Argument *name* is the name of the prover, as listed by command *why3 config list-supported-provers* and as found in file `provers-detection-data.conf`.

If the argument *shortcut* is present, it is used as the shortcut for invoking the prover.

For example, to add an Alt-Ergo executable `/home/me/bin/alt-ergo-trunk` with shortcut `new-ae`, one can type

```
why3 config add-prover Alt-Ergo /home/me/bin/alt-ergo-trunk new-ae
```

Manually added provers are stored in the configuration file under `[manual_binary]` sections as well as `[detected_binary]` ones.

6.1.2 Command `detect`

This command automatically detects the installed provers that are supported by Why3. It also creates a configuration file if none exists.

Automatically detected provers are stored in the configuration file under `[detected_binary]` sections.

6.1.3 Command `list-supported-provers`

This command lists the names of all supported provers, as used for command *why3 config add-prover*.

6.1.4 Command `show`

This command shows the expanded version of the configuration file.

6.2 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 12.3.
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 *--goal* and *--theory*. Option *--theory* applies to the previous file appearing on the command line. Option *--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 *--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 *--driver* option, or the driver of the prover selected with the *--prover* option. *why3 --list-provers* lists the known provers, the ones that appear in the configuration file.
8. If option *--prover* is given, call the selected prover on each generated task and print the results. If option *--driver* is given, print each generated task using the format specified in the selected driver.

9. Derive a validated counterexample using runtime-assertion checking, if option *--check-ce* is given and the selected prover generated a counterexample, .

6.2.1 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.

6.2.2 Options

-F <format>, **--format**=<format>
Select the given input format.

-T <theory>, **--theory**=<theory>
Focus on the given theory. If the argument is not qualified, the theory is searched in the input file.

-G <goal>, **--goal**=<goal>
Focus on the given goal. The goal is searched in the theory given by *--theory*, if any. Otherwise, it is searched in the toplevel namespace of the input file.

-a <transform>, **--apply-transform**=<transform>
Apply the given transformation to the goals.

-P <prover>, **--prover**=<prover>
Execute the given prover on the goals.

-D <driver>, **--driver**=<driver>
Output the tasks obtained by applying the given driver to the goals. This option conflicts with *--prover*.

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

--check-ce
Validate the counterexample using runtime-assertion checking. Only applicable when the prover selected by *--prover* is configured to generate a counterexample.

--rac-prover=<p>
Use prover *p* for the runtime-assertion checking during the validation of counterexamples, when term reduction is insufficient (which is always tried first). The prover *p* is the name or shortcut of a prover, with optional, comma-separated time limit and memory limit, e.g. `cvc4,2,1000`.

--rac-try-negate
Try to decide the validity of an assertion by negating the assertion and the prover answer (if any), when a prover is defined for RAC using *--rac-prover* but unable to decide the validity of the un-negated assertion.

6.2.3 Generating potential counterexamples

When the selected prover has alternative *counterexample*, the prover is instructed to generate a model, and Why3 elaborates the model into a potential counterexample. The potential counterexample associates source locations and variables to values. The generation and display of potential counterexamples is presented in details in Section 6.3.7.

6.2.4 Generating validated counterexamples

A validated counterexample can be requested using option `--check-ce`. The validated counterexample is derived by executing the relevant function using runtime assertion checking (RAC)[1]. The potential counterexample serves as an oracle for values that are not or cannot be computed in the RAC execution (e.g., arguments to the relevant function or `any`-expressions).

The validated counterexample is a trace of the RAC execution, with one of the following qualifications:

The program does not comply to the verification goal:

> The validated counterexample is the trace of an execution that resulted in the violation of an assertion.

The contracts of some function or loop are underspecified:

> The validated counterexample is the trace of an abstract execution, which resulted in the violation of an assertion. In an abstract execution, function calls and loops are not executed. Their results and assignments are instead chosen according to the contracts (function postcondition or loop invariants) by picking them from the potential counterexample.

The program does not comply to the verification goal, or the contracts of some loop or function are too weak:

> Either of the above cases.

Sorry, we don't have a good counterexample for you :(

> The RAC execution did not violate any assertions. The execution trace does not constitute a validated counterexample, and the potential counterexample is invalid, so no counterexample is shown.

The counterexample model could not be verified:

> The validated counterexample could not be derived because RAC execution was incomplete. The potential counterexample is instead shown with a warning.

6.3 The `ide` Command

The basic usage of the GUI is described by the tutorial of Section 2.2. The command-line options are the common options detailed in introduction to this chapter, plus the specific option already described for the `prove` command in Section 6.2.2.

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.

[1] The relevant function is generally only defined, when the counterexample is not generated for the VC of the complete program, for example by applying a split transformation using `--apply-transform=split_vc`.

6.3.1 Session

The session stores the transformations you performed on each verification condition, as well as the provers you ran. Such 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, 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 `why3 session 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.

6.3.2 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.

Prover list List the detected provers. Note that you can hide some provers of that list using *File* → *Preferences*, tab *Provers*.

Strategy list List the set of known strategies.

Edit Start 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.

Clean node Remove any unsuccessful proof attempt for which there is another successful proof attempt for the same goal.

Remove node Remove a proof attempt or a transformation.

Interrupt Cancel all the proof attempts currently scheduled or running.

6.3.3 Global Menus

Menu *File*

→ ***Add File to session*** Add a file to the current proof session.

→ ***Preferences*** Open a window for modifying preferred configuration parameters, see details below.

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

→ ***Save files*** Save edited source files on disk.

→ ***Save session and files*** Save 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*** Exit the GUI.

Menu *Tools*

→ ***Strategies*** Provide a set of actions that are performed on the selected goals:

→ ***Split VC*** Split the current goal into subgoals.

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

→ ***Auto level 1*** This is the same as level 0 but with a longer time limit.

→ ***Auto level 2*** This strategy 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 3*** This strategy is more elaborate than level 2. It attempts to apply a few transformations that are typically useful. It also tries the provers with a larger time limit. It also tries more provers.

A more detailed description of strategies is given in Section 12.7, 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 goals. 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*** Give access to all the known transformations.

→ ***Edit*** Start an editor on the selected task.

For automatic provers, this shows the file sent to the prover.

For interactive provers, this also makes it possible 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*** Replay all the obsolete proofs below the current node whose former state was Valid.

→ ***Replay all obsolete proofs*** Replay all the obsolete proofs below the current node.

→ ***Clean node*** Remove any unsuccessful proof attempt for which there is another successful proof attempt for the same goal.

→ ***Remove node*** Remove a proof attempt or a transformation.

→ ***Mark obsolete*** Mark all the proof as obsolete. This makes it possible to replay every proof.

→ ***Interrupt*** Cancel all the proof attempts currently scheduled or running.

→ ***Bisect*** Reduce the size of the context for the the selected proof attempt, which must be a Valid one.

→ ***Focus*** Focus the tree session view to the current node.

→ ***Unfocus*** Undo the Focus action.

→ ***Copy*** Mark the proof sub-tree for copy/past action.

→ ***Paste*** Paste the previously selected sub-tree under the current node.

Menu *View*

→ ***Enlarge font*** Select a large font.

→ ***Reduce font*** Select a smaller font.

→ ***Collapse proved goals*** Close all the rows of the tree view that are proved.

→ ***Expand all*** Expand all the rows of the tree view.

→ ***Collapse under node*** Close all the rows of the tree view under the given node that are proved.

→ ***Expand below node*** Expand the children below the current node.

→ ***Expand all below node*** Expand the whole subtree of the current node.

→ ***Go to parent node*** Move to the parent of the current node.

→ ***Go to first child*** Move to the first child of the current node.

→ ***Select next unproven goal*** Move to the next unproven goal after the current node.

Menu *Help*

→ ***Legend*** Explain the meaning of the various icons.

→ ***About*** Give some information about this software.

6.3.4 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 Fig. 2.1). 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.

6.3.5 Key shortcuts

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

- Paste: `Control-v`
- Select parent node: `Control-up`
- Select next unproven goal: `Control-down`
- Change focus to command line: `Return`
- Edit: `e`
- Replay: `r`
- Clean: `c`
- Remove: `Delete`
- Mark obsolete : `o`

6.3.6 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 *SaveClose* button will save the modified settings in Why3 configuration file, to make them permanent.

Tab *General* 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.

Tab *Appearance*

- 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 coercions 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.

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

- The default editor to use when the 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.

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

Tab *Uninstalled provers policies* presents all the decision previously taken for missing provers, as described in Section 5.2.2. You can remove any recorded decision by clicking on it.

6.3.7 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 Fig. 6.1) 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 Fig. 6.2: 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 Fig. 6.2.

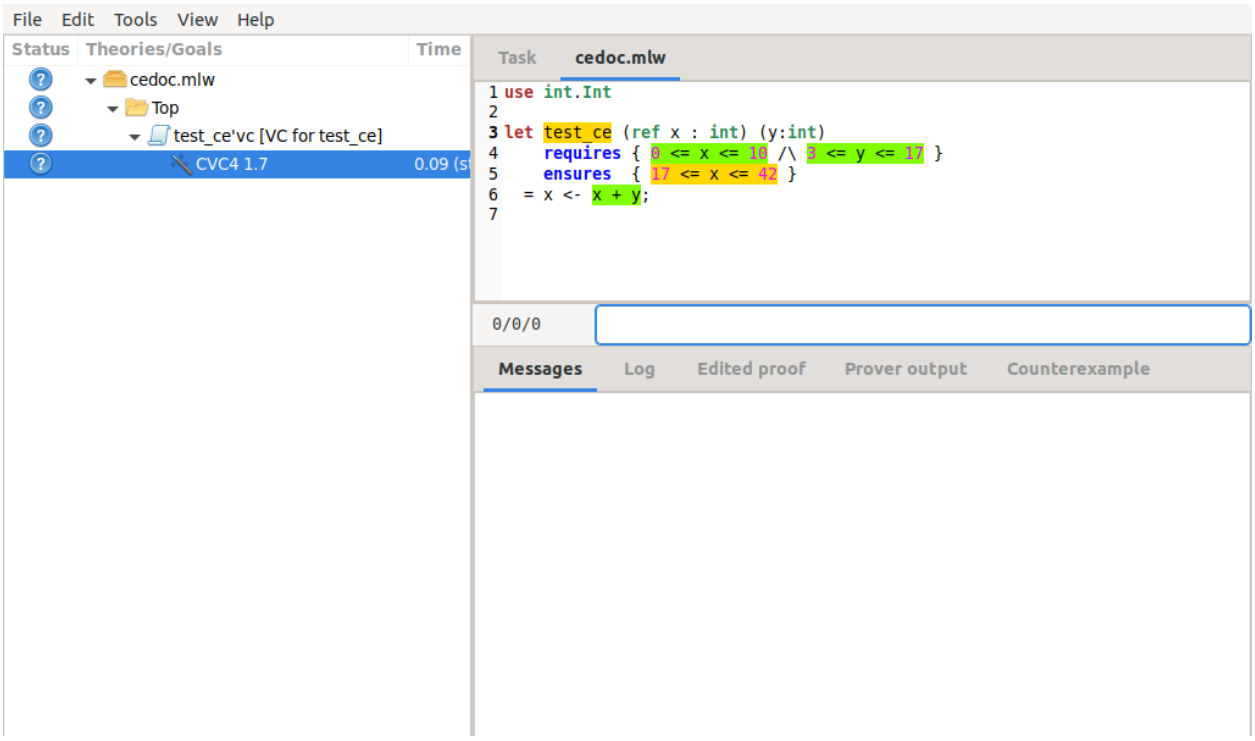

Fig. 6.1: Failing execution of CVC4

Fig. 6.2: Counterexamples display for CVC4

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.

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 `ensures` 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.

More information on the implementation of counterexamples in Why3 can be found in [HMM16] and in [DHMM18]. For the producing counterexamples using the Why3 API, see Section 4.10.

6.4 The `replay` Command

The **`why3 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 *View* → *Collapse proved goals*. In other words, when a goal, a theory, or a file is fully proved, the subtree is not shown.

6.4.1 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,
- all the goals are proved.

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

6.4.2 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
Suppress the output of the final tree view.

-q
Run quietly (no progress info).

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

--obsolete-only
Replay the proofs only if the session contains obsolete proof attempts.

--smoke-detector`[=none|top|deep]`
Try to detect if the context is self-contradicting (default: top).

--prover=<prover>
Restrict the replay to the selected provers only.

6.4.3 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.

6.5 The `session` Command

The **`why3 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:

`session info` Print information and statistics about sessions.

`session latex` Output session contents in LaTeX format.

`session html` Output session contents in HTML format.

`session update` Update session contents.

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

6.5.1 Command `info`

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

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

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

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

`--print0`
Separate the results of the options *`--provers`* and *`--edited-files`* by the null character `\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 2.

```
== 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
```

6.5.2 Command `latex`

The **`why3 session latex`** command 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>
: Set 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>
: Indicate where to produce LaTeX files (default: the session directory).

`--longtable`
: Use the `longtable` environment instead of `tabular`.

`-e` <elem>
: Produce 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.

Here are some examples of macro definitions:

```
\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}
```

6.5.3 Command `html`

The **why3 session html** 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. Fig. 6.3 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.

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	---

Fig. 6.3: HTML table produced for the HelloProof example

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``=[simpletree|table]`
: Set the style to use, among `simpletree` and `table` (default: `table`).

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

`--context`
: Add context around the generated code in order to allow direct visualization (header, css, etc.). It also adds in the output directory all the needed external files. It is incompatible with stdout output.

`--add_pp``=<suffix>,<cmd>,<out_suffix>`
: Set a specific pretty-printer for files with the given suffix. Produced files use *<out_suffix>* as suffix. *<cmd>* must contain `%i` which will be replaced by the input file and `%o` which will be replaced by the output file.

`--coqdoc`
: use the **coqdoc** command to display Coq proof scripts. This is equivalent to `--add_pp=.v,coqdoc --no-index --html -o %o %i,.html`

6.5.4 Command update

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

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

6.6 The doc Command

The **why3 doc** command 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.

6.6.1 Options

-o <dir>, **--output**=<dir>
Define the directory where to output the HTML files.

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

--no-index
Prevent the generation of an index file.

--title=<title>
Set title of the index page.

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

6.6.2 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.

6.7 The pp Command

This tool pretty-prints Why3 declarations into various forms. The kind of output is specified using the `--output` option.

```
why3 pp [--output=mlw|sexp|latex|dep] [--kind=inductive] [--prefix=<prefix>] \
  <filename> <file>[[.<Module>].<ind_type>] ...
```

--output=<output>
Set the output format, among the following:

- `mlw`: reformat WhyML source code.
- `sexp`: print the abstract syntax tree of a WhyML file (data-type from API module `Ptree`) as a S-expression (enabled only when package `ppx_sexp_conv` is available at configuration time of Why3).
- `latex`: currently can be used to print WhyML inductive definitions to LaTeX, using the `mathpartir` package.
- `dep`: display module dependencies, under the form of a digraph using the `dot` syntax from the GraphViz visualisation software.

--kind=<kind>
Set the syntactic kind to be pretty printed. Currently, the only supported kind are inductive types (`--kind=inductive`) when using the LaTeX output (`--output=latex`).

--prefix=<prefix>
Set the prefix for LaTeX commands when using `--output=latex`. The default prefix is `WHY`.

For the LaTeX output, the typesetting of variables, record fields, and functions can be configured by LaTeX commands. Dummy definitions of these commands are printed in comments and have to be defined by the user. Trailing digits and quotes are removed from the command names to reduce the number of commands.

6.8 The `execute` Command

Why3 can execute expressions in the context of a WhyML program (extension `.mlw`).

```
why3 execute [options] <file> <expr>
```

file is a WhyML file, and *expr* is a WhyML expression. Using option `--use=<M>` the definitions from module *M* are added to the context for executing *expr*. For example, the following command executes `Mod1.f 42` defined in `myfile.mlw`:

```
why3 execute myfile.mlw --use=Mod1 'f 42'
```

Upon completion of the execution, the value of the result is displayed on the standard input. Additionally, values of the global mutable variables modified by that function are displayed too.

See more details and examples of use in Section 10.1.

6.8.1 Runtime assertion checking (RAC)

The execution can be instructed using option *--rac* to check the validity of the program annotations that are encountered during the execution. This includes the validation of assertions, function contracts, and loop invariants[2].

There are two strategies to check the validity of an annotation: First, the term is reduced using the Why3 transformation `compute_in_goal`. The annotation is valid when the result of the reduction is *true* and invalid when the result is *false*. When the transformation cannot reduce the term to a trivial term, and when a RAC prover is given using option *--rac-prover*, the prover is used to verify the term.

When a program annotation is found to be wrong during the execution, the execution stops and reports the contradiction. Normally, the execution continues when an annotation cannot be checked (when the term can neither be reduced nor proven), but fails when option *–rac-fail-cannot-check* is given.

6.8.2 Options

`--use`=<Mod>
: Add the definitions from *Mod* to the execution context.

`--rac`
: Check the validity of program annotations encountered during the execution.

`--rac-prover`=<p>
: Same option as for *prove*.

`--rac-try-negate`
: Same option as for *prove*.

`--rac-fail-cannot-reduce`
: Instruct the RAC execution to fail when an annotation cannot be checked. Normally the execution continues normally when an annotation cannot be checked.

[2] RAC for function invariants aren't supported yet.

6.9 The `extract` Command

The **`why3 extract`** command can extract programs written using the WhyML language (extension `.mlw`) to some other programming language. See also Section 10.2.

The 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` in directory `<dir>`, 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 *why3 -L*, when extracting either a module or a symbol, in order to state where to look for file `f.mlw`.

-o <file|dir>
: Output extracted code to the given file (for *--flat*) or directory (for *--modular*).

-D <driver>, **--driver**=<driver>
: Use the given driver.

--flat
: Perform a flat extraction, *i.e.*, everything is extracted into a single file. This is the default behavior. If option *-o* is omitted, the result of extraction is printed to the standard output.

--modular
: Extract each module in its own, separate file. Option *-o* is mandatory; it should be given the name of an existing directory. This directory will be populated with the resulting OCaml files.

--recursive
: Recursively extract all the dependencies of the chosen entry point. This option is valid for both *--modular* and *--flat* options.

6.10 The `realize` Command

Why3 can produce skeleton files for proof assistants that, once filled, realize the given theories. See also Section 11.2.

6.11 The `wc` Command

Why3 can give some token statistics about WhyML source files.

CHAPTER

SEVEN

THE WHYML LANGUAGE REFERENCE

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

7.1 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 ("). The backslash character \, is used for escaping purposes. The following escape sequences are allowed:

- \ followed by a *new line* allows for multi-line strings. The leading spaces immediately after the new line are ignored.
- \\ and \" for the backslash and double quote respectively.
- `\n`, `\r`, and `\t` for the new line feed, carriage return, and horizontal tab character.
- `\DDD`, `\oOOO`, and `\xXX`, where `DDD` is a decimal value in the interval 0-255, `OOO` an octal value in the interval 0-377, and `XX` an hexadecimal value. Sequences of this form can be used to encode Unicode characters, in particular non printable ASCII characters.
- any other escape sequence results in a parsing error.

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

```
digit       ::=   "0" - "9"
hex_digit   ::=   "0" - "9" | "a" - "f" | "A" - "F"
oct_digit   ::=   "0" - "7"
bin_digit   ::=   "0" | "1"
integer     ::=   digit (digit | "_")*
                  | ("0x" | "0X") hex_digit (hex_digit | "_")*
                  | ("0o" | "0O") oct_digit (oct_digit | "_")*
                  | ("0b" | "0B") bin_digit (bin_digit | "_")*
real        ::=   digit+ exponent
                  | digit+ "." digit* exponent?
                  | digit* "." digit+ exponent?
                  | ("0x" | "0X") hex_digit+ h_exponent
                  | ("0x" | "0X") hex_digit+ "." hex_digit* h_exponent?
                  | ("0x" | "0X") hex_digit* "." hex_digit+ h_exponent?
exponent    ::=   ("e" | "E") ("-" | "+")? digit+
h_exponent  ::=   ("p" | "P") ("-" | "+")? digit+
char        ::=   "a" - "z" | "A" - "Z" | "0" - "9"
```

```
                  | " " | "!" | "#" | "$" | "%" | "&" | "'" | "("
                  | ")" | "*" | "+" | "," | "-" | "." | "/" | ":"
                  | ";" | "<" | "=" | ">" | "?" | "@" | "[" | "]"
                  | "^" | "_" | "`" | "\\" | "\n" | "\r" | "\t" | '\"'
                  | "\" ("0" | "1") digit digit
                  | "\" "2" ("0" - "4") digit
                  | "\" "2" "5" ("0" - "5")
                  | "\x" hex_digit hex_digit
                  | "\o" ("0" - "3" ) oct_digit oct_digit
string      ::=   '"' char* '"'
```

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

```
alpha       ::=   "a" - "z" | "A" - "Z"
suffix      ::=   (alpha | "'"* ("0" - "9" | "_")*)* "'"*
lident_nq   ::=   ("a" - "z") suffix* | "_" suffix+
uident_nq   ::=   ("A" - "Z") suffix*
ident_nq    ::=   lident_nq | uident_nq
qident      ::=   "'" ("a" - "z") suffix*
```

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.

```
lident   ::=   lident_nq ("'" alpha suffix)*
uident   ::=   lident_nq ("'" alpha suffix)*
ident    ::=   lident | uident
```

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:

```
qualifier   ::=   (uident ".")+
lqualid     ::=   qualifier? lident
uqualid     ::=   qualifier? uident
```

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:

```
op_char_1      ::=   "=" | "<" | ">" | "~"
op_char_2      ::=   "+" | "-"
op_char_3      ::=   "*" | "/" | "\" | "%"
```

```
op_char_4      ::=   "!" | "$" | "&" | "?" | "@" | "^" | "." | ":" | "|" | "#"
op_char_1234   ::=   op_char_1 | op_char_2 | op_char_3 | op_char_4
op_char_234    ::=   op_char_2 | op_char_3 | op_char_4
op_char_34     ::=   op_char_3 | op_char_4
infix_op_1     ::=   op_char_1234* op_char_1 op_char_1234* "'"*
infix_op_2     ::=   op_char_234* op_char_2 op_char_234* "'"*
infix_op_3     ::=   op_char_34* op_char_3 op_char_34* "'"*
infix_op_4     ::=   op_char_4+ "'"*
prefix_op      ::=   op_char_1234+ "'"*
tight_op       ::=   ("!" | "?") op_char_4* "'"*
```

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  ::=   "[@" ... "]"
                 | "[#" string digit+ digit+ digit+ "]"
```

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.

7.2 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.

7.3 Logical expressions

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.

7.3.1 Terms and Formulas

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 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 *term*.

```
term0       ::=   integer ; integer constant
                | real ; real constant
                | "true" | "false" ; Boolean constant
                | "()" ; empty tuple
                | string ; string constant
                | 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 ";")* ("_" "=>" term)? "|]" ; function literal
                | "[|" (term ";")+ "|]" ; function literal (domain over nat)
                | 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
                | "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
                  | ... ; (to be continued in term)
formula     ::=   term ; no distinction as far as syntax is concerned
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
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)*
```

The various constructs have the following priorities and associativities, from lowest to greatest priority:

construct	associativity
`if then else` / `let in`	–
attribute	–
cast	–
`->` / `<->` / `by` / `so`	right
\/ / ||	right
/\ / &&	right
`not`	–
infix-op level 1	right
`at` / `old`	–
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	–

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`.

Infix operators from groups 2-4 are left-associative. Infix operators from group 1 are right-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`. Note that infix symbols of level 1 include equality (=) and disequality (<>).

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.

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.

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

7.3.2 Specific syntax for collections

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.

WhyML provides a special syntax for *function literals*. The term `[|t1 => u1; ...; tn => un; _ => default|]`, where `t1, ..., tn` have some type `t` and `u1, ..., un, default` some type `u`, represents a total function of the form `fun x -> if x = t1 then u1 else if ... else if x = tn then un else default`. The default value can be omitted in which case the last value will be taken as the default value. For instance, the function literal `[|t1 => u1|]` represents the term `fun x -> if x = t1 then u1 else u1`. When the domain of the function ranges over an initial sequence of the natural numbers it is possible to write `[|t1;t2;t3|]` as a shortcut for `[|0 => t1; 1 => t2; 2 => t3|]`. Function literals cannot be empty.

7.3.3 The "at" and "old" operators

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)`.

7.3.4 Non-standard connectives

The propositional connectives in WhyML formulas are listed in *term*. 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 (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 behavior of the splitting transformations is further controlled by attributes *[@stop_split]* and *[@case_split]*. Consult the documentation of transformation *split_goal* in Section 12.6 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.

7.3.5 Conditionals, "let" bindings and pattern-matching

```
term       ::=   term0
                | "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
```

Above, 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 `t -> u`.

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,q as x)`, variable `x` is bound to the value matched by pattern `q`. 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 e`, 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.

7.4 Program expressions

The syntax of program expressions is given below. 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 (*infix_op_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
                    | string ; string constant
                    | 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 ";")* ("_" "=>" expr)? "|]" ; function literal
                  | "[|" (expr ";")+ "|]" ; function literal (domain over nat)
                  | 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
                  | 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
                  | "while" expr "do" invariant* variant? expr "done" ; while loop
                  | "for" lident "=" expr ("to" | "downto") expr "do" invariant* expr "done" ; for l
                  | "for" pattern "in" expr "with" uident ("as" lident_nq)? "do" invariant* variant?
                  | "break" lident? ; loop break
                  | "continue" lident? ; loop continue
                  | ("assert" | "assume" | "check") "{" term "}" ; assertion
                  | "raise" uqualid expr? ; exception raising
                  | "raise" "(" uqualid expr? ")"
                  | "try" expr "with" ("|" handler)+ "end" ; exception catching
                  | "(" expr ")" ; parentheses
                  | "label" uident "in" expr ; label
handler       ::= uqualid pattern? "->" expr ; exception handler
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
                    | "diverges" ; may not terminate
                    | ("reads" | "writes" | "alias") "{" "}" ; empty effect
path            ::= lqualid ("." lqualid)* ; v.field1.field2
alias           ::= path "with" path ; arg1 with result
invariant       ::= "invariant" "{" term "}" ; loop and type invariant
variant         ::= "variant" "{" variant_term ("," variant_term)* "}" ; termination variant
variant_term    ::= term ("with" lqualid)? ; variant term + WF-order
```

7.4.1 Ghost expressions

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.

7.4.2 Assignment expressions

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.

7.4.3 Auto-dereference: simplified usage of mutable variables

Some syntactic sugar is provided to ease the use of mutable variables (aka references), in such a way that the bang character is no more needed to access the value of a reference, in both logic and programs. This syntactic sugar summarized in the following table.

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.

Here is an example:

```
let ref x = 0 in while x < 100 do invariant { 0 <= x <= 100 } x <- x + 1 done
```

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` requires importing `ref.Ref` or `why3.Ref.Ref`. Operations `(:=)` and `(!)` require importing `ref.Ref`. Note that operation `(:=)` is fully subsumed by direct assignment `(<-)`.

7.4.4 Evaluation order

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

7.4.5 Referring to past program states using "at" and "old" operators

Within specifications, terms are extended with constructs `old` and `at`. Within a postcondition, `old t` refers to the value of term `t` in the pre-state. Within the scope of a code label `L`, introduced with `label L in ...`, the term `t at L` refers to the value of term `t` at the program point corresponding to `L`.

Note that `old` can be used in annotations contained in the function body as well (assertions, loop invariants), with the exact same meaning: it refers to the pre-state of the function. In particular, `old t` in a loop invariant does not refer to the program point right before the loop but to the function entry.

Whenever `old t` or `t at L` refers to a program point at which none of the variables in `t` is defined, Why3 emits a warning `this `at'/`old' operator is never used` and the operator `old`/`at` is ignored. For instance, the following code

```
let x = ref 0 in assert { old !x = !x }
```

emits a warning and is provable, as it amounts to proving *0=0*. Similarly, if `old t` or `t at L` refers to a term `t` that is immutable, Why3 emits the same warning and ignores the operator.

Caveat: Whenever the term `t` contains several variables, some of them being meaningful at the corresponding program point but others not being in scope or being immutable, there is *no warning* and the operator `old`/`at` is applied where it is defined and ignored elsewhere. This is convenient when writing terms such as `old a[i]` where `a` makes sense in the pre-state but `i` does not.

7.4.6 The "for" loop

The "for" loop of Why3 has the following general form:

```
for v=e1 to e2 do invariant { i } e3 done
```

Here, `v` is a variable identifier, that is bound by the loop statement and of type `int` ; `e1` and `e2` are program expressions of type `int`, and `e3` is an expression of type `unit`. The variable `v` may occur both in `i` and `e3`, and is not mutable. The execution of such a loop amounts to first evaluate `e1` and `e2` to values `n1` and `n2`. If `n1 >= n2` then the loop is not executed at all, otherwise it is executed iteratively for `v` taking all the values between `n1` and `n2` included.

Regarding verification conditions, one must prove that `i[v <- n1]` holds (invariant initialization) ; and that `forall n. n1 <= n <= n2 /\ i[v <- n] -> i[v <- n+1]` (invariant preservation). At loop exit, the property which is known is `i[v <- n2+1]` (notice the index `n2+1`). A special case occurs when the initial value `n1` is larger than `n2+1`: in that case the VC generator does not produce any VC to prove, the loop just acts as a no-op instruction. Yet in the case when `n1 = n2+1`, the formula `i[v <- n2+1]` is asserted and thus need to be proved as a VC.

The variant with keyword `downto` instead of `to` iterates backwards.

It is also possible for `v` to be an integer range type (see Section 7.5.3) instead of an integer.

7.4.7 The "for each" loop

The "for each" loop of Why3 has the following syntax:

```
for p in e1 with S do invariant/variant... e2 done
```

Here, `p` is a pattern, `S` is a namespace, and `e1` and `e2` are program expressions. Such a for each loop is syntactic sugar for the following:

```
let it = S.create e1 in
try
  while true do
    invariant/variant...
    let p = S.next it in
    e2
  done
with S.Done -> ()
```

That is, namespace `S` is assumed to declare at least a function `create` and a function `next`, and an exception `Done`. The latter is used to signal the end of the iteration. As shown above, the iterator is named `it`. It can be referred to within annotations. A different name can be specified, using syntax `with S as x do`.

7.4.8 Break & Continue

The `break` and `continue` statements can be used in `while`, `for` and `for-each` loops, with the expected semantics. The statements take an optional identifier which can be used to break out of nested loops. This identifier can be defined using `label` like in the following example:

```
label A in
while true do
  variant...
  while true do
    variant..
    break A (* abort the outer loop *)
  done
done
```

7.4.9 Function literals

Function literals can be written in expressions the same way as they are in terms but there are a few subtleties that one must bear in mind. First of all, if the domain of the literal is of type `t` then an equality infix operator = should exist. For instance, the literal `[|t1 => u1|]` with `t1` of type `t`, is only considered well typed if the infix operator = of type `t -> t -> bool` is visible in the current scope. This problem does not exist in terms because the equality in terms is polymorphic.

Second, the function literal expression `[|t1 => u1; t2 => u2; _ => u3|]` will be translated into the following expression:

```
let def'e = u3 in
let d'i1 = t2 in
let r'i1 = u2 in
let d'i0 = t1 in
```

(continues on next page)

(continued from previous page)

```
let r'i0 = u1 in
fun x'x -> if x'x = d'i0 then r'i0 else
           if x'x = d'i1 then r'i1 else
           def'e
```

7.4.10 The any expression

The general form of the any expression is the following.

```
any <type> <contract>
```

This expression non-deterministically evaluates to a value of the given type that satisfies the contract. For example, the code

```
let x = any int ensures { 0 <= result < 100 } in
...
```

will give to x any non-negative integer value smaller than 100.

As for contracts on functions, it is allowed to name the result or even give a pattern for it. For example the following expression returns a pair of integers which first component is smaller than the second.

```
any (int,int) returns { (a,b) -> a <= b }
```

Notice that an any expression is not supposed to have side effects nor raise exceptions, hence its contract cannot include any `writes` or `raises` clauses.

To ensure that this construction is safe, it is mandatory to show that there is always at least one possible value to return. It means that the VC generator produces a proof obligation of form

```
exists result:<type>. <post-condition>
```

In that respect, notice the difference with the construct

```
val x:<type> <contract> in x
```

which will not generate any proof obligation, meaning that the existence of the value x is taken for granted.

7.5 Modules

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

```
file            ::=  module*
module          ::=  "module" uident_nq attribute* 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?
                    | "scope" "import"? uident_nq decl* "end"
                    | "import" uident
                    | "let" "ghost"? lident_nq attribute* fun_defn
                    | "let" "rec" fun_defn
                    | "val" "ghost"? lident_nq attribute* pgm_decl
                    | "exception" lident_nq attribute* type?
type_decl       ::= lident_nq attribute* ("'" lident_nq attribute*)* type_defn
type_defn       ::= ; abstract type
                    | "=" type ; alias type
                    | "=" "|"? type_case ("|" type_case)* ; algebraic type
                    | "=" vis_mut "{" record_field (";" record_field)* "}" invariant* type_witness ;
                    | "<" "range" integer integer ">" ; range type
                    | "<" "float" integer integer ">" ; float type
type_case       ::= uident attribute* type_param*
record_field    ::= "ghost"? "mutable"? lident_nq attribute* ":" type
type_witness    ::= "by" "{" lident_nq "=" expr (";" lident_nq "=" expr)* "}"
vis_mut         ::= ("abstract" | "private")? "mutable"?
pgm_decl        ::= ":" type ; global variable
                    | param (spec* param)+ ":" type spec* ; abstract function
logic_decl      ::= function_decl
                    | predicate_decl
constant_decl   ::= lident_nq attribute* ":" type
                    | lident_nq attribute* ":" type "=" term
function_decl   ::= lident_nq attribute* type_param* ":" type
                    | lident_nq attribute* type_param* ":" type "=" term
predicate_decl  ::= lident_nq attribute* type_param*
                    | lident_nq attribute* type_param* "=" formula
inductive_decl  ::= lident_nq attribute* type_param* "=" "|"? ind_case ("|" ind_case)*
ind_case        ::= ident_nq attribute* ":" formula
imp_exp         ::= ("import" | "export")?
subst           ::= "with" "," subst_elt)+
subst_elt       ::= "type" lqualid "=" lqualid
                    | "function" lqualid "=" lqualid
                    | "predicate" lqualid "=" lqualid
                    | "scope" (uqualid | ".") "=" (uqualid | ".")
                    | "lemma" qualid
                    | "goal" qualid
tqualid         ::= uident | ident ("." ident)* "." uident
type_param      ::= "'" lident
                    | lqualid
                    | "(" lident+ ":" type ")"
                    | "(" type ("," type)* ")"
                    | "()"
```

7.5.1 Record types

A record type declaration introduces a new type, with named and typed fields, as follows:

```
type t = { a: int; b: bool }
```

Such a type can be used both in logic and programs. A new record is built using curly braces and a value for each field, such as `{ a = 42; b = true }`. If `x` is a value of type `t`, its fields are accessed using the dot notation, such as `x.a`. Each field happens to be a projection function, so that we can also write `a x`. A field can be declared `mutable`, as follows:

```
type t = { mutable a: int; b: bool }
```

A mutable field can be modified using notation `x.a <- 42`. The `writes` clause of a function contract can list mutable fields, e.g., `writes { x.a }`.

Type invariants

Invariants can be attached to record types, as follows:

```
type t = { mutable a: int; b: bool }
  invariant { b = true -> a >= 0 }
```

The semantics of type invariants is as follows. In the logic, a type invariant always holds. Consequently, it is no more possible to build a value using the curly braces (in the logic). To prevent the introduction of a logical inconsistency, Why3 generates a VC to show the existence of at least one record instance satisfying the invariant. It is named `t'vc` and has the form `exists a:int, b:bool. b = true -> a >= 0`. To ease the verification of this VC, one can provide an explicit witness using the keyword `by`, as follows:

```
type t = { mutable a: int; b: bool }
  invariant { b = true -> a >= 0 }
  by { a = 42; b = true }
```

It generates a simpler VC, where fields are instantiated accordingly.

In programs, a type invariant is assumed to hold at function entry and must be restored at function exit. In the middle, the invariant can be temporarily broken. For instance, the following function can be verified:

```
let f (x: t) = x.a <- x.a - 1; x.a <- 0
```

After the first assignment, the invariant does not necessarily hold anymore. But it is restored before function exit with the second assignment.

If the record is passed to another function, then the invariant must be reestablished (so as to honor the contract of the callee). For instance, the following function cannot be verified:

```
let f1 (x: t) = x.a <- x.a - 1; f x; x.a <- 0
```

Indeed, passing `x` to function `f` requires checking the invariant first, which does not hold in this example. Similarly, the invariant must be reestablished if the record is passed to a logical function or predicate. For instance, the following function cannot be verified:

```
predicate p (x: t) = x.b

let f2 (x: t) = x.a <- x.a - 1; assert { p x }; x.a <- 0
```

Accessing the record fields, however, does not require restoring the invariant, both in logic and programs. For instance, the following function can be verified:

```
let f2 (x: t) = x.a <- x.a - 1; assert { x.a < old x.a }; x.a <- 0
```

Indeed, the invariant may not hold after the first assignment, but the assertion is only making use of field access, so there is no need to reestablish the invariant.

Private types

A record type can be declared `private`, as follows:

```
type t = private { mutable a: int; b: bool }
```

The meaning of such a declaration is that one cannot build a record instance, neither in the logic, nor in programs. For instance, the following function cannot be defined:

```
let create () = { a = 42; b = true }
```

One cannot modify mutable fields of private types either. One may wonder what is the purpose of private types, if one cannot build values in those types. The purpose is to build interfaces, to be later refined with actual implementations (see section *Module cloning* below). Indeed, if we cannot build record instances, we can still *declare* operations that return such records. For instance, we can declare the following two functions:

```
val create (n: int) : t
  ensures { result.a = n }

val incr (x: t) : unit
  writes  { x.a }
  ensures { x.a = old x.a + 1 }
```

Later, we can *refine* type `t` with a type that is not private anymore, and then implement operations `create` and `incr`.

Private types are often used in conjunction with ghost fields, that are used to model the contents of data structures. For instance, we can conveniently model a queue containing integers as follows:

```
type queue = private { mutable ghost s: seq int }
```

If needed, we could even add invariants (e.g., the sequence `s` is sorted in a priority queue).

When a private record type only has ghost fields, one can use `abstract` as a convenient shortcut:

```
type queue = abstract { mutable s: seq int }
```

This is equivalent to the previous declaration.

Recursive record types

Record types can be recursive, e.g,

```
type t = { a: int; next: option t }
```

Recursive record types cannot have invariants, cannot have mutable fields, and cannot be private.

7.5.2 Algebraic data types

Algebraic data types combine sum and product types. A simple example of a sum type is that of an option type:

```
type maybe = No | Yes int
```

Such a declaration introduces a new type `maybe`, with two constructors `No` and `Yes`. Constructor `No` has no argument and thus can be used as a constant value. Constructor `Yes` has an argument of type `int` and thus can be used to build values such as `Yes 42`. Algebraic data types can be polymorphic, e.g.,

```
type option 'a = None | Some 'a
```

(This type is already part of Why3 standard library, in module option.Option.)

A data type can be recursive. The archetypal example is the type of polymorphic lists:

```
type list 'a = Nil | Cons 'a (list 'a)
```

(This type is already part of Why3 standard library, in module list.List.)

When a field is common to all constructors, with the same type, it can be named:

```
type t =
  | MayBe (size: int) (option int)
  | Many  (size: int) (list int)
```

Such a named field introduces a projection function. Here, we get a function `size` of type `t -> int`.

Constructor arguments can be ghost, e.g.,

```
type answer =
  | Yes (ghost int)
  | No
```

Non-uniform data types are allowed, such as the following type for random access lists:

```
type ral 'a =
  | Empty
  | Zero    (ral ('a, 'a))
  | One  'a (ral ('a, 'a))
```

Why3 supports polymorphic recursion, both in logic and programs, so that we can define and verify operations on such types.

Tuples

A tuple type is a particular case of algebraic data types, with a single constructor. A tuple type need not be declared by the user; it is generated on the fly. The syntax for a tuple type is `(type1, type2, ...)`.

Note: Record types, introduced in the previous section, also constitute a particular case of algebraic data types with a single constructor. There are differences, though. Record types may have mutable fields, invariants, or private status, while algebraic data types cannot.

7.5.3 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
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`.

7.5.4 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 [IEE08]. 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`.

7.5.5 Function declarations

let Definition of a program function, with prototype, contract, and body

val Declaration of a program function, with prototype and contract only

let function Definition of a pure (that is, side-effect free) program function which can also be used in specifications as a logical function symbol

let predicate Definition of a pure Boolean program function which can also be used in specifications as a logical predicate symbol

val function Declaration of a pure program function which can also be used in specifications as a logical function symbol

val predicate Declaration of a pure Boolean program function which can also be used in specifications as a logical predicate symbol

function Definition or declaration of a logical function symbol which can also be used as a program function in ghost code

predicate Definition or declaration of a logical predicate symbol which can also be used as a Boolean program function in ghost code

let lemma definition of a special pure program function which serves not as an actual code to execute but to prove the function's contract as a lemma: "for all values of parameters, the precondition implies the postcondition". This lemma is then added to the logical context and is made available to provers. If this "lemma-function" produces a result, the lemma is "for all values of parameters, the precondition implies the existence of a result that satisfies the postcondition". Lemma-functions are mostly used to prove some property by induction directly in Why3, without resorting to an external higher-order proof assistant.

Program functions (defined with `let` or declared with `val`) can additionally be marked `ghost`, meaning that they can only be used in the ghost code and never translated into executable code ; or `partial`, meaning that their execution can produce observable effects unaccounted by their specification, and thus they cannot be used in the ghost code.

7.5.6 Module cloning

Why3 features a mechanism to make an instance of a module, by substituting some of its declarations with other symbols. It is called *module cloning*.

Let us consider the example of a module implementing exponentiation by squaring. We want to make it as general as possible, so that we can implement it and verify it only once and then reuse it in various different contexts, e.g., with integers, floating-point numbers, matrices, etc. We start our module with the introduction of a monoid:

```
module Exp
  use int.Int
  use int.ComputerDivision

  type t

  val constant one : t

  val function mul t t : t

  axiom one_neutral: forall x. mul one x = x = mul x one

  axiom mul_assoc: forall x y z. mul x (mul y z) = mul (mul x y) z
```

Then we define a simple exponentiation function, mostly for the purpose of specification:

```
  let rec function exp (x: t) (n: int) : t
    requires { n >= 0 }
    variant  { n }
  = if n = 0 then one else mul x (exp x (n - 1))
```

In anticipation of the forthcoming verification of exponentiation by squaring, we prove two lemmas. As they require induction, we use lemma functions:

```
  let rec lemma exp_add (x: t) (n m: int)
    requires { 0 <= n /\ 0 <= m }
    variant  { n }
    ensures  { exp x (n + m) = mul (exp x n) (exp x m) }
  = if n > 0 then exp_add x (n - 1) m

  let rec lemma exp_mul (x: t) (n m: int)
    requires { 0 <= n /\ 0 <= m }
    variant  { m }
    ensures  { exp x (n * m) = exp (exp x n) m }
  = if m > 0 then exp_mul x n (m - 1)
```

Finally, we implement and verify exponentiation by squaring, which completes our module.

```
  let fast_exp (x: t) (n: int) : t
    requires { n >= 0 }
    ensures  { result = exp x n }
  = let ref p = x in
    let ref q = n in
    let ref r = one in
    while q > 0 do
```

(continues on next page)

(continued from previous page)

```
      invariant { 0 <= q }
      invariant { mul r (exp p q) = exp x n }
      variant   { q }
      if mod q 2 = 1 then r <- mul r p;
      p <- mul p p;
      q <- div q 2
    done;
    r

end
```

Note that module `Exp` mixes declared symbols (type `t`, constant `one`, function `mul`) and defined symbols (function `exp`, program function `fast_exp`).

We can now make an instance of module `Exp`, by substituting some of its declared symbols (not necessarily all of them) with some other symbols. For instance, we get exponentiation by squaring on integers by substituting `int` for type `t`, integer 1 for constant `one`, and integer multiplication for function `mul`.

```
module ExponentiationBySquaring
  use int.Int
  clone Exp with type t = int, val one = one, val mul = (*)
end
```

In a substitution such as `val one = one`, the left-hand side refers to the namespace of the module being cloned, while the right-hand side refers to the current namespace (which here contains a constant `one` of type `int`).

When a module is cloned, any axiom is automatically turned into a lemma. Thus, the `clone` command above generates two VCs, one for lemma `one_neutral` and another for lemma `mul_assoc`. If an axiom should instead remain an axiom, it should be explicitly indicated in the substitution (using `axiom mul_assoc` for instance). Why3 cannot figure out by itself whether an axiom should be turned into a lemma, so it goes for the safe path (all axioms are to be proved) by default.

Lemmas that were proved in the module being cloned (such as `exp_add` and `exp_mul` here) are not reproved. They are part of the resulting namespace, the substitution being applied to their statements. Similarly, functions that were defined in the module being cloned (such as `exp` and `fast_exp` here) are not reproved and are part of the resulting module, the substitution being applied to their argument types, return type, and definition. For instance, we get a fresh function `fast_exp` of type `int->int->int`.

We can make plenty other instances of our module `Exp`. For instance, we get Russian multiplication for free by instantiating `Exp` with zero and addition instead.

```
module Multiplication
  use int.Int
  clone Exp with type t = int, val one = zero, val mul = (+)
  goal G: exp 2 3 = 6
end
```

7.6 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.mlw`, use the syntax `file.M`, since `file.mlw` is available in Why3's default load path. For instance, the module of integers and the module of arrays indexed by integers are imported as follows:

```
use int.Int
use array.Array
```

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 mach.int.Int63
use 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 10.2).

7.6.1 Library `int`: mathematical integers

The `int` library contains several modules whose dependencies are displayed on Figure Fig. 7.1.

Fig. 7.1: Module dependencies in library `int`.

The main module is `Int` which provides basic operations like addition and multiplication, and comparisons.

The division of modulo operations are defined in other modules. They indeed come into two flavors: the module `EuclideanDivision` proposes a version where the result of the modulo is always non-negative, whereas the module `ComputerDivision` provides a version which matches the standard definition available in programming languages like C, Java or OCaml. Note that these modules do not provide any divsion or modulo operations to be used in programs. For those, you must use the module `mach.int.Int` instead, which provides these operations, including proper pre-conditions, and with the usual infix syntax `x / y` and `x % y`.

The detailed documentation of the library is available on-line at http://why3.lri.fr/stdlib/int.html

7.6.2 Library `array`: array data structure

The `array` library contains several modules whose dependencies are displayed on Figure Fig. 7.2.

Fig. 7.2: Module dependencies in library `array`.

The main module is `Array`, providing the operations for accessing and updating an array element, with respective syntax `a[i]` and `a[i] <- e`, and proper pre-conditions for the indexes. The length of an array is denoted as `a.length`. A fresh array can be created using `make l v` where `l` is the desired length and `v` is the initial value of each cell.

The detailed documentation of the library is available on-line at http://why3.lri.fr/stdlib/array.html

CHAPTER

EIGHT

THE VC GENERATORS

This chapter gives information about the various processes that generate the so-called verification conditions, VC for short, from WhyML code.

8.1 VC generation for program functions

For each program function of the form

```
let f (x$_1$: t$_1$) ... (x$_n$: t$_n$): t
  requires { Pre }
  ensures  { Post }
= body
```

a new logic goal called `f'VC` is generated. Its shape is

$$\forall x_1, \ldots, x_n, \quad Pre \Rightarrow WP(body, Post)$$

where $WP(e, Q)$ is a formula computed automatically using rules defined recursively on e.

TODO: Refer to A Pragmatic Type System for Deductive Verification

Attributes: *[@vc:divergent]* disables generation of VC for termination

Other attributes: *[@vc:annotation]*, *[@vc:sp]*, *[@vc:wp]*, *[@vc:white_box]*, *[@vc:keep_precondition]*

8.2 VC generated for type invariants

How a VC is generated for proving that a type with invariant is inhabited. Explain also the use of the *by* keyword in this context.

8.3 VC generation and lemma functions

How a VC for a program function marked as *lemma* is turned into an hypothesis for the remaining of the module.

8.4 Using strongest post-conditions

To avoid exponential explosion of WP computation, Why3 provides a mechanism similar to (TODO: cite papers here).

This can be activited locally on any program expression, by putting the *[@vc:sp]* attribute on that expression. Yet, the simplest usage is to pose this attribute on the whole body of a program function.

Show an example.

8.5 Automatic inference of loop invariants

Why3 can be executed with support for inferring loop invariants [Bau17] (see Section 5.5 for information about the compilation of Why3 with support for *infer-loop*).

There are two ways of enabling the inference of loop invariants: by passing the debug flag *infer-loop* to Why3 or by annotating `let` declarations with the *[@infer]* attribute.

Below is an example on how to invoke Why3 such that invariants are inferred for all the loops in the given file.

```
why3 ide tests/infer/incr.mlw --debug=infer-loop
```

In this case, the *Polyhedra* default domain will be used together with the default widening value of *3*. Why3 GUI will not display the inferred invariants in the source code, but the VCs corresponding to those invariants will be displayed and labeled with the `infer-loop` keyword as shown in Fig. 8.1.

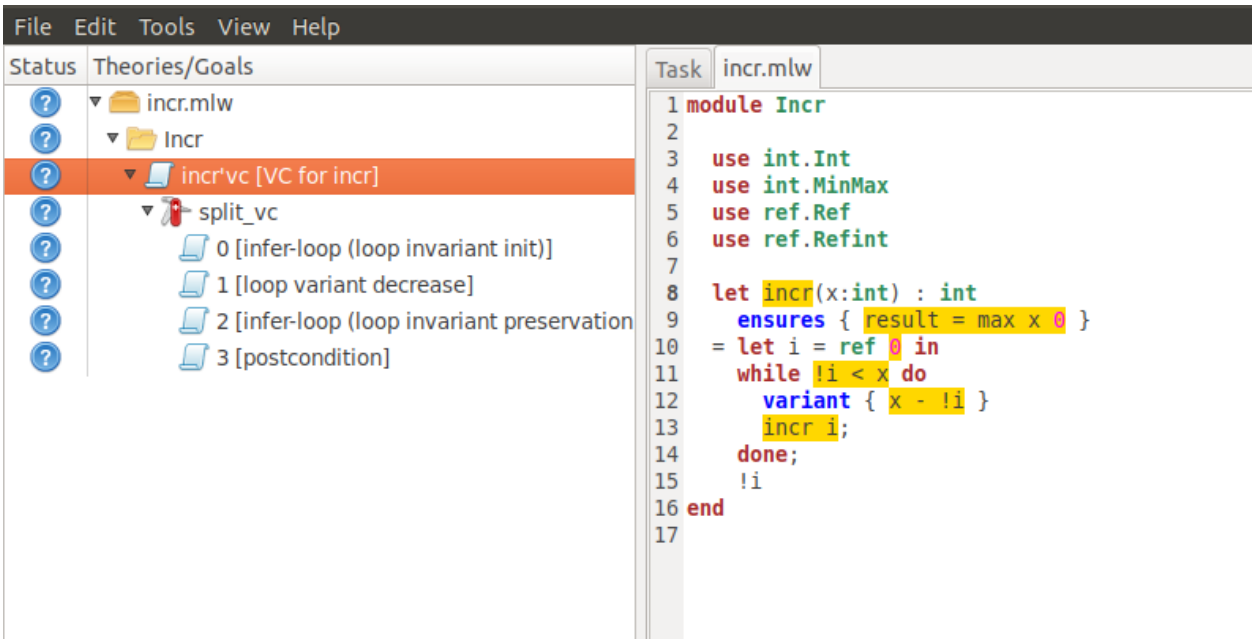

Fig. 8.1: The GUI with inferred invariants (after split).

Alternatively, attributes can be used in `let` declarations so that invariants are inferred for all the loops in that declaration. In this case, it is possible to select the desired domain and widening value. In the example below, invariants will be inferred using the *Polyhedra* domain and a widening value of *4*. These two arguments of the attribute can swapped, for instance, `[@infer:Polyhedra:4]` will produce exactly the same invariants.

```
module Incr

  use int.Int
  use int.MinMax
  use ref.Ref
  use ref.Refint

  let incr[@infer:4:Polyhedra](x:int) : int
    ensures { result = max x 0 }
  = let i = ref 0 in
    while !i < x do
      variant { x - !i }
      incr i;
    done;
    !i
end
```

There are a few debugging flags that can be passed to Why3 to output additional information about the inference of loop invariants. Flag *infer-print-cfg* will print the Control Flow Graph (CFG) used for abstract interpretation in a file with the name `inferdbg.dot`; *infer-print-ai-result* will print to the standard output the computed abstract values at each point of the CFG; *print-inferred-invs* will print the inferred invariants to the standard output (note that the displayed identifiers names might not be consistent with those in the initial program); finally, *print-domains-loop* will print for each loop the loop expression, the domain at that point, and its translation into a Why3 term.

8.5.1 Current limitations

1. Loop invariants can only be inferred for loops inside (non-recursive) `let` declarations.

CHAPTER

NINE

OTHER INPUT FORMATS

Why3 can be used to verify programs written in languages other than WhyML. Currently, Why3 supports tiny subsets of C and Python, respectively coined micro-C and micro-Python below. These were designed for teaching purposes. They come with their own specification languages, written in special comments. These input formats are described below.

Any Why3 tool (*why3 prove*, *why3 ide*, etc.) can be passed a file with a suffix `.c` or `.py`, which triggers the corresponding input format. These input formats can also be used in on-line versions of Why3, at http://why3.lri.fr/micro-C/ and http://why3.lri.fr/python/, respectively.

9.1 micro-C

Micro-C is a valid subset of ANSI C. Hence, micro-C files can be passed to a C compiler.

9.1.1 Syntax of micro-C

Logical annotations are inserted in special comments starting with `//@` or `/*@`. In the following grammar, we only use the former kind, for simplicity, but both kinds are allowed.

```
file       ::=   decl*
decl       ::=   c_include | c_function | logic_declaration
c_include  ::=   "#include" "<" file-name ">"
```

Directives `#include` are ignored during the translation to Why3. They are allowed anyway, such that a C source code using functions such as `printf` (see below) is accepted by a C compiler.

Function definition

```
c_function   ::=   return_type identifier "(" params? ")" spec* block
return_type  ::=   "void" | "int"
params       ::=   param ("," param)*
param        ::=   "int" identifier | "int" identifier "[]"
```

Function specification

```
spec  ::=   "//@" "requires" term ";"
          | "//@" "ensures" term ";"
          | "//@" "variant" term ("," term)* ";"
```

C expression

```
expr  ::=   integer-literal | string-literal
          | identifier | identifier ( "++" | "--" ) | ( "++" | "--" ) identifier
          | identifier "[" expr "]"
          | identifier "[" expr "]" ( "++" | "--")
          | ( "++" | "--") identifier "[" expr "]"
          | "-" expr | "!" expr
          | expr ( "+" | "-" | "*" | "/" | "%" | "==" | "!=" | "<" | "<=" | ">" | ">=" | "&&" | "||"
          | identifier "(" (expr ("," expr)*)? ")"
          | "scanf" "(" '"%d"' "," "&" identifier ")"
          | "(" expr ")"
```

C statement

```
stmt        ::=   ";"
                | "return" expr ";"
                | "int" identifier ";"
                | "int" identifier "[" expr "]" ";"
                | "break" ";"
                | "if" "(" expr ")" stmt
                | "if" "(" expr ")" stmt "else" stmt
                | "while" "(" expr ")" loop_body
                | "for" "(" expr_stmt ";" expr ";" expr_stmt ")" loop_body
                | expr_stmt ";"
                | block
                | "//@" "label" identifier ";"
                | "//@" ( "assert" | "assume" | "check" ) term ";"
block       ::=   "{" stmt* "}"
expr_stmt   ::=   "int" identifier "=" expr
                | identifier assignop expr
                | identifier "[" expr "]" assignop expr
                | expr
assignop    ::=   "=" | "+=" | "-=" | "*=" | "/="
loop_body   ::=   loop_annot* stmt
                | "{" loop_annot* stmt* "}"
loop_annot  ::=   "//@" "invariant" term ";"
                | "//@" "variant" term ("," term)* ";"
```

Note that the syntax for loop bodies allows the loop annotations to be placed either before the block or right at the beginning of the block.

Logic declarations

```
logic_declaration  ::=   "//@" "function" "int" identifier "(" params ")" ";"
                       | "//@" "function" "int" identifier "(" params ")" "=" term ";"
                       | "//@" "predicate" identifier "(" params ")" ";"
                       | "//@" "predicate" identifier "(" params ")" "=" term ";"
                       | "//@" "axiom" identifier ":" term ";"
                       | "//@" "lemma" identifier ":" term ";"
                       | "//@" "goal" identifier ":" term ";"
```

Logic functions are limited to a return type `int`.

Logical term

```
term    ::=   identifier
            | integer-literal
            | "true"
            | "false"
            | "(" term ")"
            | term "[" term "]"
            | term "[" term "<-" term "]"
            | "!" term
            | "old" "(" term ")"
            | "at" "(" term "," identifier ")"
            | "-" term
            | term ( "->" | "<->" | "||" | "&&" ) term
            | term ( "==" | "!=" | "<" | "<=" | ">" | ">=" ) term
            | term ( "+" | "-" | "*" | "/" | "% ) term
            | "if" term "then" term "else term
            | "let" identifier "=" term "in" term
            | ( "forall" | "exists" ) binder ("," binder)* "." term
            | identifier "(" (term ("," term)*)? ")"
binder  ::=   identifier
            | identifier "[]"
```

9.1.2 Built-in functions and predicates

C code

- `scanf` is limited to the syntax `scanf("%d", &x)`.
- `printf` is limited to `printf(string-literal, expr1, ..., exprn)`. The string literal should contain exactly `n` occurrences of `%d` (not checked by Why3).
- `rand()` returns a pseudo-random integer in the range `0` to `RAND_MAX` inclusive.

Logic

- `int length(int a[])` returns the length of array `a`.
- `int occurrence(int v, int a[])` returns the number of occurrences of the value `v` in array `a`.

9.2 micro-Python

Micro-Python is a valid subset of Python 3. Hence, micro-Python files can be passed to a Python interpreter.

9.2.1 Syntax of micro-Python

Notation: In the grammar of micro-Python given below, special symbols `NEWLINE`, `INDENT`, and `DEDENT` mark an end of line, the beginning of a new indentation block, and its end, respectively.

Logical annotations are inserted in special comments starting with `#@`.

```
file       ::=   decl*
decl       ::=   py_import | py_function | stmt | logic_declaration
py_import  ::=   "from" identifier "import" identifier ("," identifier)* NEWLINE
```

Directives `import` are ignored during the translation to Why3. They are allowed anyway, such that a Python source code using functions such as `randint` is accepted by a Python interpreter (see below).

Function definition

```
py_function  ::=   "def" identifier "(" params? ")" ":" NEWLINE INDENT spec* stmt* DEDENT
params       ::=   identifier ("," identifier)*
```

Function specification

```
spec  ::=   "#@" "requires" term NEWLINE
          | "#@" "ensures" term NEWLINE
          | "#@" "variant" term ("," term)* NEWLINE
```

Python expression

```
expr  ::=   "None" | "True" | "False" | integer-literal | string-literal
          | identifier
          | identifier "[" expr "]"
          | "-" expr | "not" expr
          | expr ( "+" | "-" | "*" | "//" | "%" | "==" | "!=" | "<" | "<=" | ">" | ">=" | "and" | "or
          | identifier "(" (expr ("," expr)*)? ")"
          | "[" (expr ("," expr)*)? "]"
          | "(" expr ")"
```

Python statement

```
stmt          ::=   simple_stmt NEWLINE
                  | "if" expr ":" suite else_branch
                  | "while" expr ":" loop_body
                  | "for" identifier "in" expr ":" loop_body
else_branch   ::=   /* nothing */
                  | "else:" suite
                  | "elif" expr ":" suite else_branch
suite         ::=   simple_stmt NEWLINE
                  | NEWLINE INDENT stmt stmt* DEDENT
simple_stmt   ::=   expr
                  | "return" expr
                  | identifier "=" expr
                  | identifier "[" expr "]" "=" expr
                  | "break"
                  | "#@" "label" identifier
                  | "#@" ( "assert" | "assume" | "check" ) term
loop_body     ::=   simple_stmt NEWLINE
                  | NEWLINE INDENT loop_annot* stmt stmt* DEDENT
loop_annot    ::=   "#@" "invariant" term NEWLINE
                  | "#@" "variant" term ("," term)* NEWLINE
```

Logic declaration

```
logic_declaration  ::=   "#@" "function" identifier "(" params ")" NEWLINE
                       | "#@" "predicate" identifier "(" params ")" NEWLINE
```

Note that logic functions and predicates cannot be given definitions. Yet, they can be axiomatized, using toplevel `assume` statements.

Logical term

```
term  ::=   identifier
          | integer-literal
          | "None"
          | "True"
          | "False"
          | "(" term ")"
          | term "[" term "]"
          | term "[" term "<-" term "]"
          | "not" term
          | "old" "(" term ")"
          | "at" "(" term "," identifier ")"
          | "-" term
          | term ( "->" | "<->" | "or" | "and" ) term
          | term ( "==" | "!=" | "<" | "<=" | ">" | ">=" ) term
          | term ( "+" | "-" | "*" | "//" | "%" ) term
          | "if" term "then" term "else term
          | "let" identifier "=" term "in" term
```

```
| ( "forall" | "exists" ) identifier ("," identifier)* "." term
| identifier "(" (term ("," term)*)? ")"
```

9.2.2 Built-in functions and predicates

Python code

- `len(l)` returns the length of list `l`.
- `int(input())` reads an integer from standard input.
- `range(l, u)` returns the list of integers from `l` inclusive to `u` exclusive. In particular, `for x in range(l, u):` is supported.
- `randint(l, u)` returns a pseudo-random integer in the range `l` to `u` inclusive.

Logic

- `len(l)` returns the length of list `l`.
- `occurrence(v, l)` returns the number of occurrences of the value `v` in list `l`.

9.2.3 Limitations

Python lists are modeled as arrays, whose size cannot be modified.

9.3 MLCFG: function bodies on the style of control-flow graphs

The MLCFG language is an experimental extension of the regular WhyML language, in which the body of program functions can be coded using labelled blocks and goto statements. MLCFG can be used to encode algorithms which are presented in an unstructured fashion. It can also be used as a target for encoding unstructured programming languages in Why3, for example assembly code.

9.3.1 Syntax of the MLCFG language

The MLCFG syntax is an extension of the regular WhyML syntax. Every WhyML declaration is allowed, plus an additional declaration of program function of the following form, introduced by keywords `let cfg`:

```
let cfg f (x$_1$: t$_1$) ... (x$_n$: t$_n$): t
  requires { Pre }
  ensures  { Post }
=
 var y$_1$: u$_1$;
 ...
 var y$_k$: u$_k$;
 { instructions;*terminator* }
 L$_1$ { instructions$_1$;*terminator*:sub:1 }
 ...
 L$_j$ { instructions$_j$;*terminator*:sub:j }
```

It defines a program function *f*, with the usual syntax for its contract. The difference is the body, which is made of a sequence of declarations of mutable variables with their types, an initial block, composed of a zero or more instructions followed by a terminator, and a sequence of other blocks, each denoted by a label ($L_1 \ldots L_j$ above). The instructions are semi-colon separated sequences of regular WhyML expressions of type `unit`, excluding `return` or `absurd` expressions or code invariants:

- a code invariant: `invariant I { t }` where *I* is a name and *t* a predicate. It is similar to an assert expression, meaning that *t* must hold when execution reaches this statement. Additionally, it acts as a cut in the generation of VC, similarly to a loop invariant. See example below.

Each block is ended by one of the following terminators:

- a `goto` statement: `goto L` where *L* is one of the labels of the other blocks. It instructs to continue execution at the given block.
- a `switch` statement, of the form

  ```
  switch (e)
  | pat$_1$ -> terminator$_1$
  ...
  | pat$_k$ -> terminator$_k$
  end
  ```

- a `return` statement: `return *expr*`
- an `absurd` statement: indicating that this block should be unreachable.

The extension of syntax is described by the following rules.

```
file          ::=   module*
module        ::=   "module" ident decl* "end"
decl          ::=   "let" "cfg" cfg_fundef
                    | "let" "rec" "cfg" cfg_fundef ("with" cfg_fundef)*
                    | "scope" ident decl* "end"
cfg_fundef    ::=   ident binder+ :  type spec "=" vardecl* "{" block "}" labelblock*
vardecl       ::=   "var" ident* ":" type ";" | "ghost" "var" ident* ":" type ";"
block         ::=   (instruction ";")* terminator
labelblock    ::=   ident "{" block "}"
instruction   ::=   expr
                    | "invariant" ident "{" term "}"
terminator    ::=
                    | "return" expr
                    | "absurd"
                    | "goto" ident
                    | "switch" "(" expr ")" switch_case* "end"
switch_case   ::=   "|" pattern "->" terminator
```

9.3.2 An example

The following example is directly inspired from the documentation of the ANSI C Specification Language (See [BFM+18], Section 2.4.2 Loop invariants, Example 2.27). It is itself inspired from the first example of Knuth's MIX language, for which formal proofs were first investigated by J.-C. Filliâtre in 2007 ([Filliatre07]), and also revisited by T.-M.-T. Nguyen in her PhD thesis in 2012 ([Ngu12], Section 9.5 Translation from a CFG to Why, page 115).

This example aims at computing the maximum value of an array of integers. Its code in C is given below.

```
/*@ requires n >= 0 && \valid(a,0,n);
  @ ensures \forall integer j ; 0 <= j < n ==> \result >= a[j]);
  @*/
int max_array(int a[], int n) {
  int m, i = 0;
  goto L;
  do {
    if (a[i] > m) { L: m = a[i]; }
    /*@ invariant
      @   0 <= i < n && \forall integer j ; 0 <= j <= i ==> m >= a[j]);
      @*/
    i++;
  }
  while (i < n);
  return m;
}
```

The code can be viewed as a control-flow graph as shown in Fig. 9.1.

Below is a version of this code in the Why3-CFG language, where label `L` corresponds to node `L`, label `L1` to node `invariant`, label `L2` to node `do`.

```
let cfg max_array (a:array int) : (max: int, ghost ind:int)
  requires { length a > 0 }
  ensures  { 0 <= ind < length a }
  ensures  { forall j. 0 <= j < length a -> a[ind] >= a[j] }
=
var i m: int;
ghost var ind: int;
{
  i <- 0;
  goto L
}
L {
  m <- a[i];
  ind <- i;
  goto L1
}
L1 {
  invariant i_bounds   { 0 <= i < length a };
  invariant ind_bounds { 0 <= ind < length a };
  invariant m_and_ind  { m = a[ind] };
  invariant m_is_max   { forall j. 0 <= j <= i -> m >= a[j] };
                         (* yes, j <= i, not j < i ! *)
  i <- i + 1;
  switch (i < length a)
```

(continues on next page)

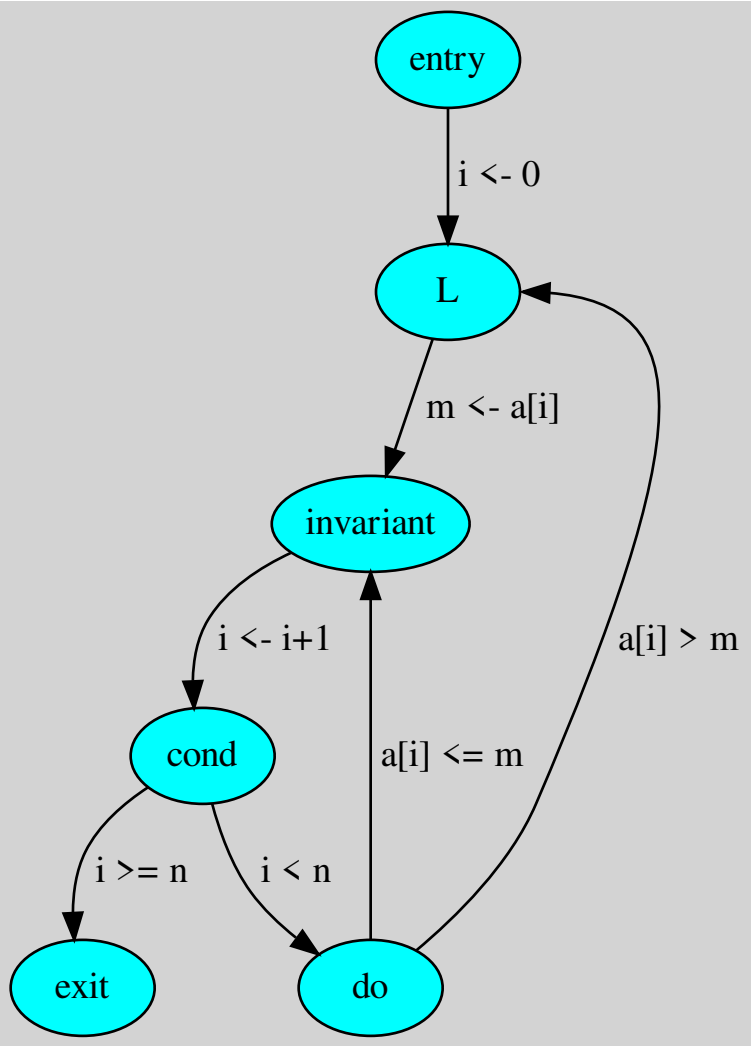

Fig. 9.1: Control-flow graph of the `max_array` function.

(continued from previous page)

```
  | True  -> goto L2
  | False -> return (m, ind)
  end
}
L2 {
  switch (a[i] > m)
  | True  -> goto L
  | False -> goto L1
  end
}
```

The consecutive invariants act as a single cut in the generation of VCs.

9.3.3 Error messages

The translation from the CFG language to regular WhyML code may raise the following errors.

- "cycle without invariant": in order to perform the translation, any cycle on the control-flow graph must contain at least one `invariant` clause. It corresponds to the idea that any loop must contain a loop invariant.
- "cycle without invariant (starting from I)": same error as above, except that the cycle was not reachable from the start of the function body, but from the other `invariant` clause named I.
- "label L not found for goto": there is a `goto` instruction to a non-existent label.
- "unreachable code after goto": any code occurring after a `goto` statement is unreachable and is not allowed.
- "unsupported: trailing code after switch": see limitations below.

9.3.4 Current limitations

- There is no way to prove termination.
- New keywords `cfg`, `goto`, `switch`, and `var` cannot be used as regular identifiers anymore.
- Trailing code after `switch` is not supported. In principle, it should be possible to have a `switch` with type `unit` and to transfer the execution to the instructions after the `switch` for branches not containing `goto`. This is not yet supported. A workaround is to place the trailing instructions in another block and pose an extra `goto` to this block in all the branches that do not end with a `goto`.
- Conditional statements `if e then i1 else i2` are not yet supported, but can be simulated with `switch (e) | True -> i1 | False -> i2 end`.

CHAPTER

TEN

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 of Section 3.2 that computes the maximum and the sum of an array of integers.

Let us assume it is contained in a file `maxsum.mlw`.

10.1 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 --use=MaxAndSum 'test ()'
result: (int, int) = (45, 10)
globals:
```

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

10.2 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`.

10.2.1 Examples

We illustrate different ways of using the *extract* command through some examples.

Consider the program of Section 3.6.

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 the *extract* command 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 =
```

(continues on next page)

(continued from previous page)

```
  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).

10.2.2 Extraction of Functors

WhyML and OCaml are both dialects of the ML-family, sharing many syntactic and semantics traits. Yet their module systems differ significantly. A WhyML program is a list of modules, a module is a list of top-level declarations, and declarations can be organized within *scopes*, the WhyML unit for namespaces management. In particular, there is no support for sub-modules in Why3, nor a dedicated syntactic construction for functors. The latter are represented, instead, as modules containing only abstract symbols [FilliatreP20]. One must follow exactly this programming pattern when it comes to extract an OCaml functor from a Why3 proof. Let us consider the following (excerpt) of a WhyML module implementing binary search trees:

```
module BST
  scope Make
    scope Ord
      type t
      val compare : t -> t -> int
    end

    type elt = Ord.t

    type t = E | N t elt t

    use int.Int

    let rec insert (x: elt) (t: t)
    = match t with
      | E -> N E x E
      | N l y r ->
          if Ord.compare x y > 0 then N l y (insert x r)
          else N (insert x l) y r
      end
  end
end
```

For the sake of simplicity, we omit here behavioral specification. Assuming the above example is contained in a file named `bst.mlw`, one can readily extract it into OCaml, as follows:

```
> why3 extract -D ocaml64 bst.mlw --modular -o .
```

This produces the following functorial implementation:

```
module Make (Ord: sig type t
  val compare : t -> t -> Z.t end) =
struct
  type elt = Ord.t

  type t =
```

(continues on next page)

(continued from previous page)

```
  | E
  | N of t * Ord.t * t

  let rec insert (x: Ord.t) (t: t) : t =
    match t with
    | E -> N (E, x, E)
    | N (l, y, r) ->
        if Z.gt (Ord.compare x y) Z.zero
        then N (l, y, insert x r)
        else N (insert x l, y, r)
end
```

The extracted code features the functor `Make` parameterized with a module containing the abstract type `t` and function `compare`. This is similar to the OCaml standard library when it comes to data structures parameterized by an order relation, *e.g.*, the `Set` and `Map` modules.

From the result of the extraction, one understands that scope `Make` is turned into a functor, while the nested scope `Ord` is extracted as the functor argument. In summary, for a WhyML implementation of the form

```
module M
  scope A
    scope X ... end
    scope Y ... end
    scope Z ... end
  end
  ...
end
```

contained in file `f.mlw`, the Why3 extraction engine produces the following OCaml code:

```
module A (X: ...) (Y: ...) (Z: ...) = struct
  ...
end
```

and prints it into file `f__M.ml`. In order for functor extraction to succeed, scopes `X`, `Y`, and `Z` can only contain non-defined programming symbols, *i.e.*, abstract type declarations, function signatures, and exception declarations. If ever a scope mixes non-defined and defined symbols, or if there is no surrounding scope such as `Make`, the extraction will complain about the presence of non-defined symbols that cannot be extracted.

It is worth noting that extraction of functors only works for *modular* extraction (*i.e.* with command-line option *--modular*).

10.2.3 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

ELEVEN

INTERACTIVE PROOF ASSISTANTS

11.1 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*.

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 *File* → *Preferences*, under the *Editors* tab.

11.2 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.

11.2.1 Generating a realization

Generating the skeleton for a theory is done by passing to the `realize` command and 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 `why3 -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`.

11.2.2 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 *realized_theory* 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.

11.2.3 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 *realized_theory* meta 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 *why3 --extra-config* option.

11.3 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.

11.4 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 the *Tools* → *Edit* action in the Why3 IDE.

11.4.1 Installation

You need version Isabelle2018 or Isabelle2019. Former or later versions are not supported. We assume below that your version is 2019, please replace 2019 by 2018 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/Isabelle2019/etc/components
```

or the system-wide file

```
<Isabelle install dir>/etc/components
```

11.4.2 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 interactive session, any use of *Tools* → *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 `Control-w`.

11.4.3 Using Isabelle 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
```

11.4.4 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.

11.5 PVS

11.5.1 Installation

You need version 6.0.

11.5.2 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).

11.5.3 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

TWELVE

TECHNICAL INFORMATIONS

12.1 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 format CDATA #IMPLIED>
<!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 (path*, (result|undone|internalfailure|unedited))>
```

(continues on next page)

(continued from previous page)

```
<!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|failure|highfailure)
↪#REQUIRED>
<!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>
```

12.2 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.

12.3 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 *`why3 --config`* option,
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.

12.3.1 Extra Configuration Files

In addition to the main configuration file, Why3 commands accept the option *why3 --extra-config* to read one or more files containing additional configuration option. It allows the user to pass extra declarations in prover drivers, as illustrated in Section 12.5, including declarations for realizations, as illustrated in Section 11.2.

12.4 Drivers for External Provers

Drivers for external provers are readable files from directory `drivers`. They describe how Why3 should interact with external provers.

Files with `.drv` extension represent drivers that might be associated to a specific solver in the `why3.conf` configuration file (see Section 12.3 for more information); files with `.gen` extension are intended to be imported by other drivers; finally, files with `.aux` extension are automatically generated from the main `Makefile`.

The most important drivers dependencies are shown in the following figures: Fig. 12.1 shows the drivers files for SMT solvers, Fig. 12.2 for TPTP solvers, Fig. 12.3 for Coq, Fig. 12.4 for Isabelle/HOL, and Fig. 12.5 for PVS.

12.5 Adding extra drivers for user theories

It is possible for the users to augment the system drivers with extra information for their own declared theories. The processus is described by the following example.

First, we define a new theory in a file `bvmisc.mlw`, containing

```
theory T
use bv.BV8
use bv.BV16
function lsb BV16.t : BV8.t (** least significant bits *)
function msb BV16.t : BV8.t (** most significant bits *)
end
```

For such a theory, it is a good idea to declare specific translation rules for provers that have a built-in bit-vector support, such as Z3 and CVC4 in this example. To do so, we write a extension driver file, `my.drv`, containing

```
theory bvmisc.T
syntax function lsb "((_ extract 7 0) %1)"
syntax function msb "((_ extract 15 8) %1)"
end
```

Now to tell Why3 that we would like this driver extension when calling Z3 or CVC4, we write an extra configuration file, `my.conf`, containing

```
[prover_modifiers]
prover = "CVC4"
driver = "my.drv"
```

(continues on next page)

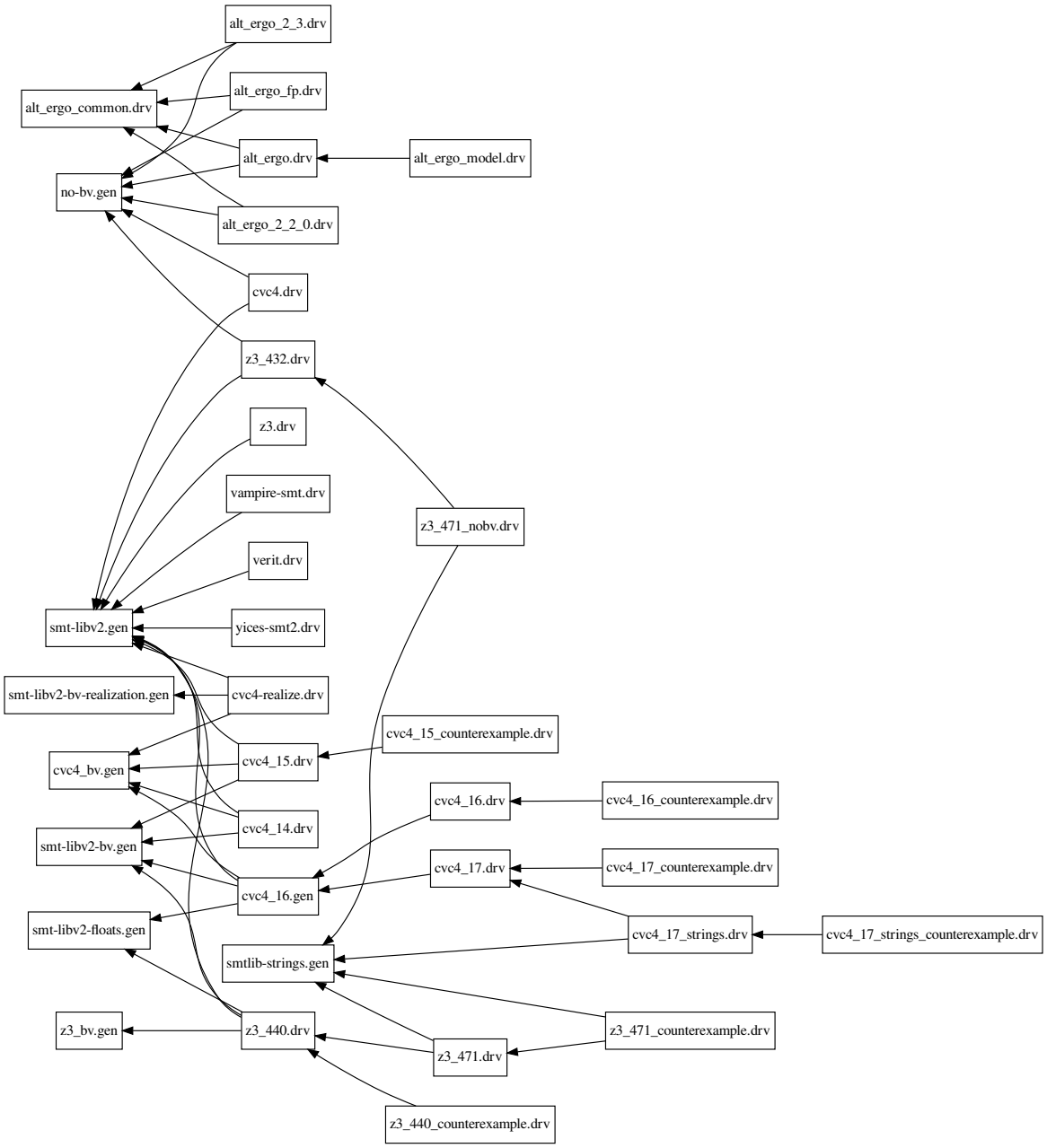

Fig. 12.1: Driver dependencies for SMT solvers

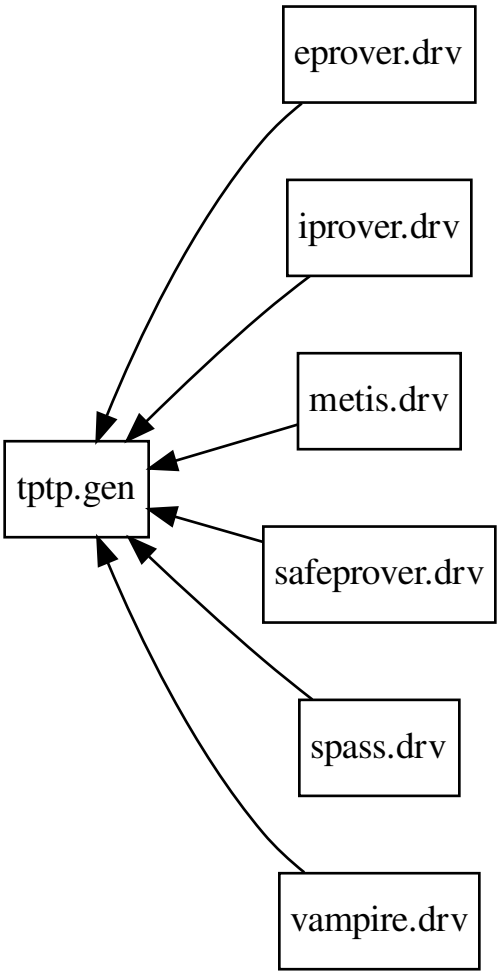

Fig. 12.2: Driver dependencies for TPTP solvers

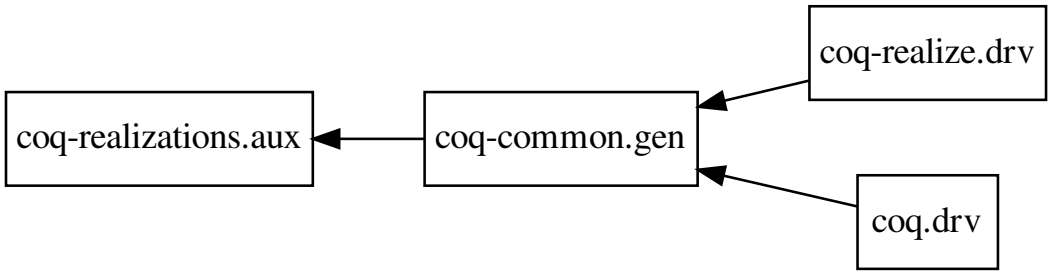

Fig. 12.3: Driver dependencies for Coq

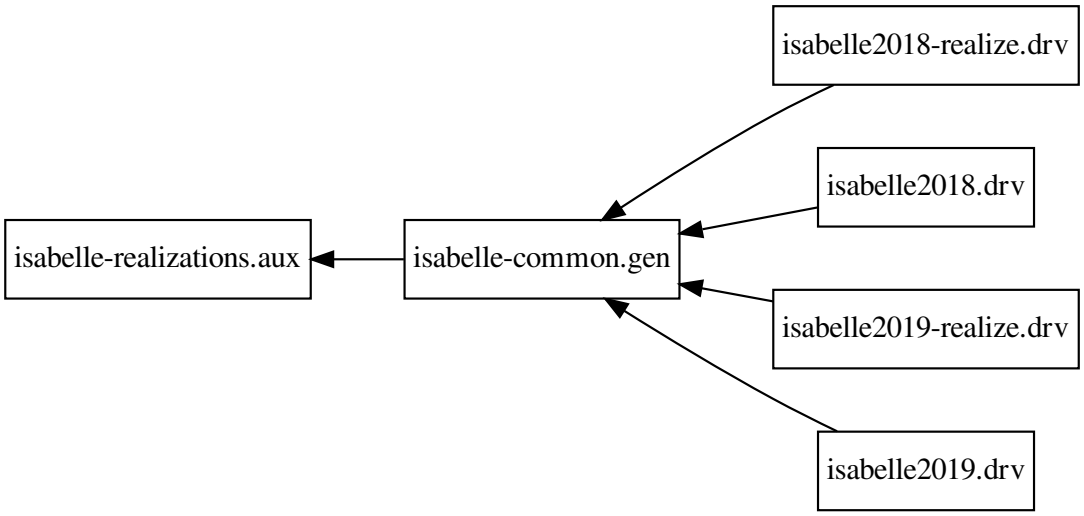

Fig. 12.4: Driver dependencies for Isabelle/HOL

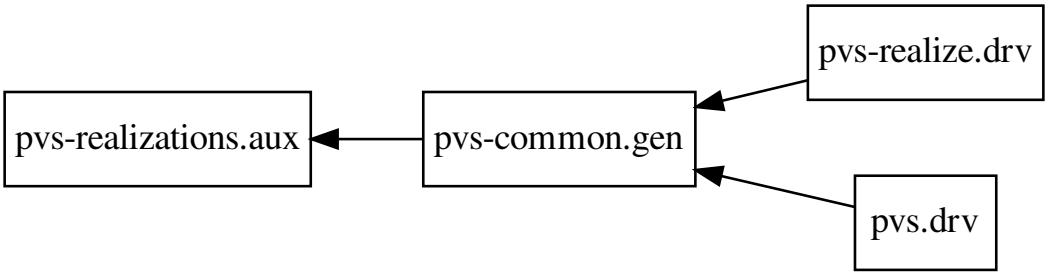

Fig. 12.5: Driver dependencies for PVS

(continued from previous page)

```
[prover_modifiers]
prover = "Z3"
driver = "my.drv"
```

Finally, to make the whole thing work, we have to call any Why3 command with the additional option *why3 --extra-config*, such as

```
why3 --extra-config=my.conf prove myfile.mlw
```

12.6 Transformations

This section documents the available transformations. Note that the set of available transformations in your own installation is given by *why3 --list-transforms*.

apply

Apply an hypothesis to the goal of the task using a *modus ponens* rule. The hypothesis should be an implication whose conclusion can be matched with the goal. The intuitive behavior of *apply* can be translated as follows. Given $\Gamma, h : f_1 \to f_2 \vdash G : f_2$, `apply h` generates a new task $\Gamma, h : f_1 \to f_2 \vdash G : f_1$.

In practice, the transformation also manages to instantiate some variables with the appropriate terms.

For example, applying the transformation `apply zero_is_even` on the following goal

```
predicate is_even int
predicate is_zero int
axiom zero_is_even: forall x: int. is_zero x -> is_even x
goal G: is_even 0
```

produces the following goal:

```
predicate is_even int
predicate is_zero int
```

(continues on next page)

(continued from previous page)

```
axiom zero_is_even: forall x:int. is_zero x -> is_even x
goal G: is_zero 0
```

The transformation first matched the goal against the hypothesis and instantiated `x` with `0`. It then applied the *modus ponens* rule to generate the new goal.

This transformation helps automated provers when they do not know which hypothesis to use in order to prove a goal.

`apply with`

Variant of *`apply`* intended to be used in contexts where the latter cannot infer what terms to use for variables given in the applied hypothesis.

For example, applying the transformation `apply transitivity` on the following goal

```
axiom ac: a = c
axiom cb: c = b
axiom transitivity : forall x y z:int. x = y -> y = z -> x = z
goal G1 : a = b
```

raises the following error:

```
apply: Unable to infer arguments (try using "with") for: y
```

It means that the tool is not able to infer the right term to instantiate symbol `y`. In our case, the user knows that the term `c` should work. So, it can be specified as follows: `apply transitivity with c`

This generates two goals which are easily provable with hypotheses `ac` and `cb`.

When multiple variables are needed, they should be provided as a list in the transformation. For the sake of the example, we complicate the `transitivity` hypothesis:

```
axiom t : forall x y z k:int. k = k -> x = y -> y = z -> x = z
```

A value can be provided for `k` as follows: `apply t with c,0`.

`assert`

Create an intermediate subgoal. This is comparable to `assert` written in WhyML code. Here, the intent is only to help provers by specifying one key argument of the reasoning they should use.

Example: From a goal of the form $\Gamma \vdash G$, the transformation `assert (n = 0)` produces the following two tasks: $\Gamma \vdash h : n = 0$ and $\Gamma, h : n = 0 \vdash G$. This effectively adds `h` as an intermediate goal to prove.

`assert as`

Same as *`assert`*, except that a name can be given to the new hypothesis. Example: `assert (x = 0) as x0`.

`case`

Split a goal into two subgoal, using the *excluded middle* on a given formula. On the task $\Gamma \vdash G$, the transformation `case f` produces two tasks: $\Gamma, h : f \vdash G$ and $\Gamma, h : \neg f \vdash G$.

For example, applying `case (x = 0)` on the following goals

```
constant x : int
constant y : int
goal G: if x = 0 then y = 2 else y = 3
```

produces the following goals:

```
constant x : int
constant y : int
axiom h : x = 0
goal G : if x = 0 then y = 2 else y = 3
```

```
constant x : int
constant y : int
axiom h : not x = 0
goal G : if x = 0 then y = 2 else y = 3
```

The intent is again to simplify the job of automated provers by giving them a key argument of the reasoning behind the proof of a subgoal.

case as
Same as *case*, except that a name can be given to the new hypothesis. Example: `case (x = 0) as x0`.

clear_but
Remove all the hypotheses except those specified in the arguments. This is useful when a prover fails to find relevant hypotheses in a very large context. Example: `clear_but h23,h25`.

compute_hyp
Apply the transformation *compute_in_goal* on the given hypothesis.

compute_hyp_specified
Apply the transformation *compute_specified* on the given hypothesis.

compute_in_goal
Aggressively apply all known computation/simplification 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*.

The computations performed by this transformation 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.

compute_specified
Same as *compute_in_goal*, but perform rewriting using only built-in operators and user-provided rules.

cut
Same as *assert*, but the order of generated subgoals is reversed.

destruct
Eliminate the head symbol of a hypothesis.

For example, applying `destruct h` on the following goal

```
constant p1 : bool
predicate p2 int
axiom h : p1 = True /\ (forall x:int. p2 x)
goal G : p2 0
```

removes the logical connective `/\` and produces

```
constant p1 : bool
predicate p2 int
axiom h1 : p1 = True
axiom h : forall x:int. p2 x
goal G : p2 0
```

destruct can be applied on the following constructions:

- `false`, `true`,
- `/\`, `\/`, `->`, `not`,
- `exists`,
- `if ... then ... else ...`,
- `match ... with ... end`,
- (in)equality on constructors of the same type.

destruct_rec
Recursively call *destruct* on the generated hypotheses. The recursion on implication and `match` stops after the first occurence of a different symbol.

For example, applying `destruct_rec H` on the following goal

```
predicate a
predicate b
predicate c
axiom H : (a -> b) /\ (b /\ c)
goal G : false
```

does not destruct the implication symbol because it occurs as a subterm of an already destructed symbol. This restriction applies only to implication and `match` Other symbols are destructed recursively. Thus, in the generated task, the second `/\` is simplified:

```
predicate a
predicate b
predicate c
axiom H2 : a -> b
axiom H1 : b
axiom H: c
goal G : false
```

destruct_term
Destruct an expression according to the type of the expression. The transformation produces all possible outcomes of a destruction of the algebraic type.

For example, applying `destruct_term a` on the following goal

```
type t = A | B int
constant a : t
goal G : a = A
```

produces the following two goals:

```
type t = A | B int
constant a : t
constant x : int
axiom h : a = B x
goal G : a = A
```

```
type t = A | B int
constant a : t
axiom h : a = A
goal G : a = A
```

The term was destructed according to all the possible outcomes in the type. Note that, during destruction, a new constant `x` has been introduced for the argument of constructor `B`.

destruct_term using
Same as *destruct_term*, except that names can be given to the constants that were generated.

destruct_term_subst
Same as *destruct_term*, except that it also substitutes the created term.

eliminate_algebraic
Replace algebraic data types by first-order definitions [Pas09].

eliminate_builtin
Remove definitions of symbols that are declared as builtin in the driver, with a "syntax" rule.

eliminate_definition_func
Replace all function definitions with axioms.

eliminate_definition_pred
Replace all predicate definitions with axioms.

eliminate_definition
Apply both *eliminate_definition_func* and *eliminate_definition_pred*.

eliminate_if
Apply both *eliminate_if_term* and *eliminate_if_fmla*.

eliminate_if_fmla
Replace formulas of the form `if f1 then f2 else f3` by an equivalent formula using implications and other connectives.

eliminate_if_term
Replace 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_inductive
Replace inductive predicates by (incomplete) axiomatic definitions, construction axioms and an inversion axiom.

eliminate_let
Apply both *eliminate_let_fmla* and *eliminate_let_term*.

eliminate_let_fmla
Eliminate `let` by substitution, at the predicate level.

eliminate_let_term
Eliminate `let` by substitution, at the term level.

eliminate_literal

eliminate_mutual_recursion
Replace mutually recursive definitions with axioms.

eliminate_recursion
Replace all recursive definitions with axioms.

encoding_smt
Encode polymorphic types into monomorphic types [BCCL08].

encoding_tptp
Encode theories into unsorted logic.

exists
Instantiate an existential quantification with a witness.

For example, applying `exists 0` on the following goal

```
goal G : exists x:int. x = 0
```

instantiates the symbol `x` with `0`. Thus, the goal becomes

```
goal G : 0 = 0
```

hide
Hide a given term, by creating a new constant equal to the term and then replacing all occurences of the term in the context by this constant.

For example, applying `hide t (1 + 1)` on the goal

```
constant y : int
axiom h : forall x:int. x = (1 + 1)
goal G : (y - (1 + 1)) = ((1 + 1) - (1 + 1))
```

replaces all the occurrences of `(1 + 1)` with `t`, which gives the following goal:

```
constant y : int
constant t : int
axiom H : t = (1 + 1)
axiom h : forall x:int. x = t
goal G : (y - t) = (t - t)
```

hide_and_clear
First apply *hide* and then remove the equality between the hidden term and the introduced constant. This means that the hidden term completely disappears and cannot be recovered.

induction
Perform a reasoning by induction for the current goal.

For example, applying `induction n` on the following goal

```
constant n : int
predicate p int
predicate p1 int
axiom h : p1 n
goal G : p n
```

performs an induction on `n` starting at `0`. The goal for the base case is

```
constant n : int
predicate p int
predicate p1 int
axiom h : p1 n
axiom Init : n <= 0
goal G : p n
```

while the recursive case is

```
constant n : int
predicate p int
predicate p1 int
axiom h : p1 n
axiom Init : 0 < n
axiom Hrec : forall n1:int. n1 < n -> p1 n1 -> p n1
goal G : p n
```

induction from
Same as *induction*, but it starts the induction from a given integer instead of `0`.

induction_arg_pr
Apply *induction_pr* on the given hypothesis/goal symbol.

induction_arg_ty_lex
Apply *induction_ty_lex* on the given symbol.

induction_pr

induction_ty_lex
Perform 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, 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.

inline_trivial
Expand and remove definitions of the form

```
function  f x1 ... xn = g e1 ... ek
predicate p x1 ... xn = q e1 ... ek
```

when each `ei` is either a ground term or one of the `xj`, and each `x1 ... xn` occurs at most once in all the `ei`.

The attribute *[@inline:trivial]* can be used to tag functions, so that the transformation forcefully expands them (not using the conditions above). This can be used to ensure that some specific functions are inlined for automatic provers (*inline_trivial* is used in many drivers).

inline_goal
Expand all outermost symbols of the goal that have a non-recursive definition.

inline_all
Expand all non-recursive definitions.

instantiate
Generate a new hypothesis with quantified variables replaced by the given terms.

For example, applying `instantiate h 0, 1` on the following goal

```
predicate p int
axiom h : forall x:int, y:int. x <= y -> p x /\ p y
goal G : p 0
```

generates a new hypothesis:

```
predicate p int
axiom h : forall x:int, y:int. x <= y -> p x /\ p y
axiom Hinst : 0 <= 1 -> p 0 /\ p 1
goal G : p 0
```

This is used to help automatic provers that are generally better at working on instantiated hypothesis.

inst_rem
Apply *instantiate* then remove the original instantiated hypothesis.

introduce_premises
Move antecedents of implications and universal quantifications of the goal into the premises of the task.

intros
Introduce universal quantifiers in the context.

For example, applying `intros n, m` on the following goal

```
predicate p int int int
goal G : forall x:int, y:int, z:int. p x y z
```

produces the following goal:

```
predicate p int int int
constant n : int
constant m : int
goal G : forall z:int. p n m z
```

intros_n
Same as *intros*, but stops after the nth quantified variable or premise.

For example, applying `intros_n 2` on the following goal

```
predicate p int int int
goal G : forall x:int, y:int, z:int. p x y z
```

produces the following goal:

```
predicate p int int int
constant x : int
constant y : int
goal G : forall z:int. p x y z
```

inversion_arg_pr
Apply *inversion_pr* on the given hypothesis/goal symbol.

inversion_pr

left
Remove the right part of the head disjunction of the goal.

For example, applying `left` on the following goal

```
constant x : int
goal G : x = 0 \/ x = 1
```

produces the following goal:

```
constant x : int
goal G : x = 0
```

pose
Add a new constant equal to the given term.

For example, applying `pose t (x + 2)` to the following goal

```
constant x : int
goal G : true
```

produces the following goal:

```
constant x : int
constant t : int
axiom H : t = (x + 2)
goal G : true
```

remove
Remove a hypothesis from the context.

For example, applying `remove h` on the following goal

```
axiom h : true
goal G : true
```

produces the following goal:

```
goal G : true
```

replace
Replace a term with another one in a hypothesis or in the goal. This generates a new goal which asks for the proof of the equality.

For example, applying `replace (y + 1) (x + 2) in h` on the following goal

```
constant x : int
constant y : int
axiom h : x >= (y + 1)
goal G : true
```

produces the following two goals:

```
constant x : int
constant y : int
axiom h : x >= (x + 2)
goal G : true
```

```
constant x : int
constant y : int
axiom h : x >= (y + 1)
goal G : (y + 1) = (x + 2)
```

It can be seen as the combination of *assert* and *rewrite*.

revert

Opposite of *intros*. It takes hypotheses/constants and quantifies them in the goal.

For example, applying `revert x` on the following goal

```
constant x : int
constant y : int
axiom h : x = y
goal G : true
```

produces the following goal:

```
constant y : int
goal G : forall x:int. x = y -> true
```

rewrite

Rewrite using the given equality hypothesis.

For example, applying `rewrite eq` on the following goal

```
function a int : bool
function b int : bool
constant y : int
axiom eq : forall x:int. not x = 0 -> a x = b x
goal G : a y = True
```

produces the following goal:

```
function a int : bool
function b int : bool
constant y : int
axiom eq : forall x:int. not x = 0 -> a x = b x
goal G : b y = True
```

It also produces a goal for the premise of the equality hypothesis (as would *apply*):

```
function a int : bool
function b int : bool
constant y : int
axiom eq : forall x:int. not x = 0 -> a x = b x
goal G : not y = 0
```

rewrite with

Variant of *rewrite* intended to be used in contexts where the latter cannot infer what terms to use for the variables of the given hypotheses (see also *apply with*).

For example, the transformation `rewrite eq with 0` can be applied to the following goal:

```
function a int : bool
function b int : bool
constant y : int
axiom eq : forall x:int, z:int. z = 0 -> not x = 0 -> a x = b x
goal G : a y = True
```

Here, a value is provided for the symbol `z`. This leads to the following three goals. One is the rewritten one, while the other two are for the premises of the equality hypothesis.

```
function a int : bool
function b int : bool
constant y : int
axiom eq : forall x:int, z:int. z = 0 -> not x = 0 -> a x = b x
goal G : b y = True
```

```
function a int : bool
function b int : bool
constant y : int
axiom eq : forall x:int, z:int. z = 0 -> not x = 0 -> a x = b x
goal G : 0 = 0
```

```
function a int : bool
function b int : bool
constant y : int
axiom eq : forall x:int, z:int. z = 0 -> not x = 0 -> a x = b x
goal G : not y = 0
```

rewrite_list
Variant of *rewrite* that allows simultaneous rewriting in a list of hypothesis/goals.

right
Remove the left part of the head disjunction of the goal.

For example, applying `right` on the following goal

```
constant x : int
goal G : x = 0 \/ x = 1
```

produces the following goal:

```
constant x : int
goal G : x = 1
```

simplify_array
Automatically rewrite the task using the lemma `Select_eq` of theory `map.Map`.

simplify_formula
Reduce trivial equalities `t=t` to true and then simplify propositional structure: removes `true`, `false`, simplifies `f /\ f` to `f`, etc.

simplify_formula_and_task
Apply *simplify_formula* and remove the goal if it is equivalent to true.

simplify_recursive_definition
Reduce 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
Simplify 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
Apply *simplify_trivial_quantification*, but only in the goal.

split_all
Perform both *split_premise* and *split_goal*.

split_all_full
Perform both *split_premise* and *split_goal_full*.

split_goal
Change 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.

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. Examples:

- 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`.

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_goal_full
Behave similarly to *split_goal*, but also convert the goal to conjunctive normal form. The number of subgoals generated may be exponential in the size of the initial goal.

split_intro
Perform both *split_goal* and *introduce_premises*.

split_premise
Replace 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
Behave similarly to *split_premise*, but also convert the axioms to conjunctive normal form. The number of axioms generated per initial axiom may be exponential in the size of the initial axiom.

subst
Substitute a given constant using an equality found in the context. The constant is removed.

For example, when applying `subst x` on the following goal

```
constant x : int
constant y : int
constant z : int
axiom h : x = y + 1
axiom h1 : z = (x + y)
goal G : x = z
```

the transformation first finds the hypothesis `h` that can be used to rewrite `x`. Then, it replaces every occurrences of `x` with `y + 1`. Finally, it removes `h` and `x`. The resulting goal is as follows:

```
constant y : int
constant z : int
axiom h1 : z = ((y + 1) + y)
goal G : (y + 1) = z
```

This transformation is used to make the task more easily readable by a human during debugging. This transformation should not help automatic provers at all as they generally implement substitution rules in their logic.

unfold

Unfold the definition of a logical symbol in the given hypothesis.

For example, applying `unfold p` on the following goal

```
predicate p (x:int) = x <= 22
axiom h : forall x:int. p x -> p (x - 1)
goal G : p 21
```

produces the following goal:

```
predicate p (x:int) = x <= 22
axiom h : forall x:int. p x -> p (x - 1)
goal G : 21 <= 22
```

One can also unfold in the hypothesis, using `unfold p in h`, which gives the following goal:

```
predicate p (x:int) = x <= 22
axiom h : forall x:int. x <= 22 -> (x - 1) <= 22
goal G : 21 <= 22
```

use_th

Import a theory inside the current context. This is used, in some rare case, to reduced the size of the context in other goals, since importing a theory in the WhyML code would the theory available in all goals whereas the theory is only needed in one specific goal.

For example, applying `use_th int.Int` on the following goal

```
predicate p int
goal G : p 5
```

imports the `Int` theory. So, one is able to use the addition over integers, e.g., `replace 5 (2 + 3)`.

Any lemma appearing in the imported theory can also be used.

Note that axioms are also imported. So, this transformation should be used with care. We recommend to use only theories that do not contain any axiom because this transformation could easily make the context inconsistent.

12.7 Proof Strategies

As seen in Section 6.3, 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` unconditionally jumps to label *lab*.
- `lab:` declares the label *lab*. The default label `exit` stops 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 2.3.0, CVC4 1.7 and Z3 4.8.4 were detected by the *why3 config* command.

Split_VC is bound to the 1-line strategy

```
t split_vc exit
```

Auto_level_0 is bound to

```
c Z3,4.8.4, 1 1000
c Alt-Ergo,2.3.0, 1 1000
c CVC4,1.7, 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

```
c Z3,4.8.4, 5 1000
c Alt-Ergo,2.3.0, 5 1000
c CVC4,1.7, 5 1000
```

Same as Auto_level_0 but with 5 seconds instead of 1.

Auto_level_2 is bound to

```
start:
c Z3,4.8.4, 1 1000
c Alt-Ergo,2.3.0, 1 1000
c CVC4,1.7, 1 1000
t split_vc start
c Z3,4.8.4, 10 4000
c Alt-Ergo,2.3.0, 10 4000
c CVC4,1.7, 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 larger limits.

Auto level 3 is bound to

```
start:
c Z3,4.8.4, 1 1000
c Eprover,2.0, 1 1000
c Spass,3.7, 1 1000
c Alt-Ergo,2.3.0, 1 1000
c CVC4,1.7, 1 1000
t split_vc start
c Z3,4.8.4, 5 2000
c Eprover,2.0, 5 2000
```

(continues on next page)

(continued from previous page)

```
c Spass,3.7, 5 2000
c Alt-Ergo,2.3.0, 5 2000
c CVC4,1.7, 5 2000
t introduce_premises afterintro
afterintro:
t inline_goal afterinline
g trylongertime
afterinline:
t split_all_full start
trylongertime:
c Z3,4.8.4, 30 4000
c Eprover,2.0, 30 4000
c Spass,3.7, 30 4000
c Alt-Ergo,2.3.0, 30 4000
c CVC4,1.7, 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.

12.8 WhyML Attributes

case_split

induction

infer

inline:trivial

model_trace

rewrite

stop_split

vc:annotation

vc:divergent

vc:keep_precondition

vc:sp

vc:wp

vc:white_box

12.9 Why3 Metas

compute_max_steps

keep:literal

realized_theory

rewrite

rewrite_def

12.10 Debug Flags

infer-loop

infer-print-ai-result

infer-print-cfg

print-inferred-invs

print-domains-loop

stack_trace

12.11 Updating Syntax Error Messages

Here is the developer's recipe to update a syntax error message. We do it on the following illustrative example.

```
function let int : int
```

If such a file is passed to Why3, one obtains:

```
File "bench/parsing/bad/498_function.mlw", line 1, characters 9-12:
syntax error
```

The recipe given here provides a way to produce a more informative message. It is based on handcrafted error messages provided by the Menhir parser generator.

The first step is to call **menhir** with option `--interpret-error` while giving as input the erroneous input, under the form of a sequence of tokens as generated by `src/parser/lexer.mll`.

```
$ echo "decl_eof: FUNCTION LET" | menhir --base src/parser/parser --interpret-error src/
↪parser/parser_common.mly src/parser/parser.mly
decl_eof: FUNCTION LET
##
## Ends in an error in state: 1113.
##
## pure_decl -> FUNCTION . function_decl list(with_logic_decl) [ VAL USE TYPE THEORY␣
↪SCOPE PREDICATE MODULE META LET LEMMA INDUCTIVE IMPORT GOAL FUNCTION EXCEPTION EOF END␣
↪CONSTANT COINDUCTIVE CLONE AXIOM ]
##
## The known suffix of the stack is as follows:
```

(continues on next page)

(continued from previous page)

```
## FUNCTION
##

<YOUR SYNTAX ERROR MESSAGE HERE>
```

The text returned by that command should be appended to `src/parser/handcrafted.messages`, with of course an appropriate error message, such as this one:

```
expected function name must be a non-reserved uncapitalized identifier (token LIDENT_NQ),
↪ found "$0"
```

CHAPTER

THIRTEEN

RELEASE NOTES

13.1 Release Notes 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 `(<-)`.

13.2 Release Notes for version 1.0: syntax changes w.r.t. 0.88

The syntax of WhyML programs changed in release 1.0. The following table summarizes the changes.

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`	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 ... }`

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 1.0 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 1.0 it should be written as

```
let f <parameters> <contract> = [@vc:divergent] <body>
```

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 1.0, 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 1.0 it should be written as

```
val x:t ensures { P } in x
```

13.3 Release Notes for version 0.80: syntax changes w.r.t. 0.73

The syntax of WhyML programs changed in release 0.80. The following table summarizes the changes.

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 }`

`abstract e { Q }`	`abstract e ensures { Q }`

13.4 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
   - 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

[IEE08] IEEE standard for floating-point arithmetic. 2008. doi:10.1109/IEEESTD.2008.4610935.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *23rd International Conference on Computer Aided Verification*, 171–177. Snowbird, UT, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2032305.2032319.

[Bau17] Lucas Baudin. Deductive verification with the help of abstract interpretation. Technical Report, Univ Paris-Sud, November 2017.

[BFM+18] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. 2018. https://frama-c.com/acsl.html. URL: https://frama-c.com/acsl.html.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer-Verlag, 2004. doi:10.1007/978-3-662-07964-5.

[BCCL08] 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, 1–5. 2008. URL: http://www.lri.fr/~conchon/publis/conchon-smt08.pdf, doi:10.1145/1512464.1512466.

[CC08] Sylvain Conchon and Évelyne Contejean. The Alt-Ergo automatic theorem prover. 2008. URL: http://alt-ergo.lri.fr/.

[DHMM18] 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. URL: https://hal.inria.fr/hal-01802488.

[Filliatre07] Jean-Christophe Filliâtre. Formal Verification of MIX Programs. In *Journées en l'honneur de Donald E. Knuth*. Bordeaux, France, October 2007. http://knuth07.labri.fr/exposes.php. URL: http://www.lri.fr/~filliatr/publis/verifmix.pdf.

[FilliatreP20] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in why3. In Tiziana Margaria and Bernhard Steffen, editors, *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 12476 of Lecture Notes in Computer Science, 122–142. Rhodes, Greece, October 2020. Springer. URL: http://why3.lri.fr/isola-2020/.

[FM07] 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, 173–177. Berlin, Germany, July 2007. URL: https://hal.inria.fr/inria-00270820v1, doi:10.1007/978-3-540-73368-3_21.

[HMM16] 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, 215–233. Vienna, Austria, 2016. URL: https://hal.inria.fr/hal-01314885, doi:10.1007/978-3-319-41591-8_15.

[Ngu12] Thi Minh Tuyen Nguyen. *Taking architecture and compiler into account in formal proofs of numerical programs*. Thèse de Doctorat, Université Paris-Sud, June 2012. URL: http://tel.archives-ouvertes.fr/tel-00710193.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[Pas09] Andrei Paskevich. Algebraic types and pattern matching in the logical language of the Why verification platform. Technical Report 7128, INRIA, 2009. URL: http://hal.inria.fr/inria-00439232.

[SM10] Natarajan Shankar and Peter Mueller. Verified Software: Theories, Tools and Experiments (VSTTE'10). Software Verification Competition. August 2010. URL: http://www.macs.hw.ac.uk/vstte10/Competition.html.

INDEX

Symbols

&&, 88
-C
    why3 command line option, 61
-D
    why3-extract command line option, 81
    why3-prove command line option, 64
-F
    why3-prove command line option, 64
-G
    why3-prove command line option, 64
-L
    why3 command line option, 61
-P
    why3-prove command line option, 64
-T
    why3-prove command line option, 64
--add_pp
    why3-session-html command line option, 77
--apply-transform
    why3-prove command line option, 64
--check-ce
    why3-prove command line option, 64
--config
    why3 command line option, 61
--context
    why3-session-html command line option, 77
--coqdoc
    why3-session-html command line option, 77
--debug
    why3 command line option, 62
--debug-all
    why3 command line option, 62
--driver
    why3-extract command line option, 81
    why3-prove command line option, 64
--edited-files
    why3-session-info command line option, 74
--extra-config
    why3 command line option, 61
--extra-expl-prefix
    why3-prove command line option, 64
--flat
    why3-extract command line option, 81
--force
    why3-replay command line option, 73
--format
    why3-prove command line option, 64
--goal
    why3-prove command line option, 64
--help
    why3 command line option, 62
--index
    why3-doc command line option, 78
--kind
    why3-pp command line option, 79
--library
    why3 command line option, 61
--list-debug-flags
    why3 command line option, 61
--list-formats
    why3 command line option, 62
--list-metas
    why3 command line option, 62
--list-printers
    why3 command line option, 61
--list-provers
    why3 command line option, 62
--list-transforms
    why3 command line option, 61
--longtable
    why3-session-latex command line option, 76
--modular
    why3-extract command line option, 81
--no-index
    why3-doc command line option, 78
--obsolete-only
    why3-replay command line option, 73
--output
    why3-doc command line option, 78
    why3-pp command line option, 79
--prefix
    why3-pp command line option, 79

--print0
    why3-session-info command line option, 74
--prover
    why3-prove command line option, 64
    why3-replay command line option, 73
--provers
    why3-session-info command line option, 74
--rac
    why3-execute command line option, 80
--rac-fail-cannot-reduce
    why3-execute command line option, 80
--rac-prover
    why3-execute command line option, 80
    why3-prove command line option, 64
--rac-try-negate
    why3-execute command line option, 80
    why3-prove command line option, 64
--recursive
    why3-extract command line option, 81
--rename-file
    why3-session-update command line option, 78
--smoke-detector
    why3-replay command line option, 73
--stats
    why3-session-info command line option, 74
--stdlib-url
    why3-doc command line option, 78
--style
    why3-session-html command line option, 77
    why3-session-latex command line option, 75
--theory
    why3-prove command line option, 64
--title
    why3-doc command line option, 78
--use
    why3-execute command line option, 80
-a
    why3-prove command line option, 64
-e
    why3-session-latex command line option, 76
-o
    why3-doc command line option, 78
    why3-extract command line option, 81
    why3-session-html command line option, 77
    why3-session-latex command line option, 75
-q
    why3-replay command line option, 73
-s
    why3-replay command line option, 73
||, 88

A

abstract type, 99
algebraic data type, 100
any expression, 96
apply
    transformation, 139
apply with
    transformation, 140
assert
    transformation, 140
assert as
    transformation, 140
assignment expressions, 92
at, 94
    syntax, 88
attribute
    case_split, 156
    induction, 156
    infer, 156
    inline:trivial, 156
    model_trace, 156
    rewrite, 156
    stop_split, 156
    vc:annotation, 156
    vc:divergent, 156
    vc:keep_precondition, 156
    vc:sp, 156
    vc:white_box, 156
    vc:wp, 156
auto-dereference, 18, 92, 159

B

bracket
    syntax, 88
break, 95
by, 88

C

case
    transformation, 140
case as
    transformation, 141
case_split
    attribute, 156
CFG, 116
clear_but
    transformation, 141
clone, 102
collections
    syntax; function literals, 88, 95
command
    config, 62
    config add-prover, 62

config detect, 63
config list-supported-provers, 63
config show, 63
doc, 78
execute, 79
extract, 80
ide, 65
pp, 79
prove, 63
realize, 81
replay, 72
session, 74
session html, 76
session info, 74
session latex, 75
session update, 77
wc, 81
compute_hyp
transformation, 141
compute_hyp_specified
transformation, 141
compute_in_goal
transformation, 141
compute_max_steps
meta, 157
compute_specified
transformation, 142
conditionals
syntax, 89
config
command, 62
config add-prover
command, 62
config detect
command, 63
config list-supported-provers
command, 63
config show
command, 63
configuration file, 62, 128
continue, 95
Coq proof assistant, 128
cut
transformation, 142

D

debug flag
infer-loop, 157
infer-print-ai-result, 157
infer-print-cfg, 157
print-domains-loop, 157
print-inferred-invs, 157
stack_trace, 157
destruct
transformation, 142
destruct_rec
transformation, 142
destruct_term
transformation, 143
destruct_term using
transformation, 143
destruct_term_subst
transformation, 143
detached, 66
doc
command, 78
driver file, 127

E

Einstein's problem, 13
eliminate_algebraic
transformation, 143
eliminate_builtin
transformation, 144
eliminate_definition
transformation, 144
eliminate_definition_func
transformation, 144
eliminate_definition_pred
transformation, 144
eliminate_if
transformation, 144
eliminate_if_fmla
transformation, 144
eliminate_if_term
transformation, 144
eliminate_inductive
transformation, 144
eliminate_let
transformation, 144
eliminate_let_fmla
transformation, 144
eliminate_let_term
transformation, 144
eliminate_literal
transformation, 144
eliminate_mutual_recursion
transformation, 144
eliminate_recursion
transformation, 144
encoding_smt
transformation, 144
encoding_tptp
transformation, 144
environment variable
PATH, 57
WHY3CONFIG, 62, 134
evaluation order, 93

execute
    command, 79
exists
    transformation, 144
extract
    command, 80

F

for each loop, 94
for loop, 94, 95

G

ghost expressions, 92

H

hide
    transformation, 144
hide_and_clear
    transformation, 145

I

ide
    command, 65
induction
    attribute, 156
    transformation, 145
induction from
    transformation, 145
induction_arg_pr
    transformation, 145
induction_arg_ty_lex
    transformation, 145
induction_pr
    transformation, 146
induction_ty_lex
    transformation, 146
infer
    attribute, 156
infer-loop
    debug flag, 157
infer-print-ai-result
    debug flag, 157
infer-print-cfg
    debug flag, 157
inline:trivial
    attribute, 156
inline_all
    transformation, 146
inline_goal
    transformation, 146
inline_trivial
    transformation, 146
inst_rem
    transformation, 147
instantiate
    transformation, 146
introduce_premises
    transformation, 147
intros
    transformation, 147
intros_n
    transformation, 147
invariant
    for each loop, 94
    for loop, 94
    type, 98
inversion_arg_pr
    transformation, 147
inversion_pr
    transformation, 147
Isabelle proof assistant, 129

K

keep:literal
    meta, 157

L

label, 94
left
    transformation, 147
let
    syntax, 89

M

mapping type, 85
meta
    compute_max_steps, 157
    keep:literal, 157
    realized_theory, 157
    rewrite, 157
    rewrite_def, 157
micro-C, 111
model_trace
    attribute, 156
module cloning, 102

O

obsolete, 8, 66, 73
old, 94
    syntax, 88

P

past program states, 94
PATH, 57
pattern-matching
    syntax, 89

pose
    transformation, 148
pp
    command, 79
print-domains-loop
    debug flag, 157
print-inferred-invs
    debug flag, 157
private type, 99
proof assistant
    Coq, 128
    Isabelle, 129
    PVS, 130
prove
    command, 63
PVS proof assistant, 130
Python, 114

R

range type, 101
realize
    command, 81
realized_theory
    meta, 157
record type, 97
reference, 92
remove
    transformation, 148
replace
    transformation, 148
replay
    command, 72
revert
    transformation, 148
rewrite
    attribute, 156
    meta, 157
    transformation, 149
rewrite with
    transformation, 149
rewrite_def
    meta, 157
rewrite_list
    transformation, 150
right
    transformation, 150

S

session
    command, 74
session html
    command, 76
session info
    command, 74
session latex
    command, 75
session update
    command, 77
simplify_array
    transformation, 150
simplify_formula
    transformation, 150
simplify_formula_and_task
    transformation, 150
simplify_recursive_definition
    transformation, 150
simplify_trivial_quantification
    transformation, 150
simplify_trivial_quantification_in_goal
    transformation, 151
snapshot type, 85
so, 88
split_all
    transformation, 151
split_all_full
    transformation, 151
split_goal
    transformation, 151
split_goal_full
    transformation, 153
split_intro
    transformation, 153
split_premise
    transformation, 153
split_premise_full
    transformation, 153
stack_trace
    debug flag, 157
standard library, 104
stop_split
    attribute, 156
subst
    transformation, 153

T

transformation
    apply, 139
    apply with, 140
    assert, 140
    assert as, 140
    case, 140
    case as, 141
    clear_but, 141
    compute_hyp, 141
    compute_hyp_specified, 141
    compute_in_goal, 141
    compute_specified, 142
    cut, 142

destruct, 142
destruct_rec, 142
destruct_term, 143
destruct_term using, 143
destruct_term_subst, 143
eliminate_algebraic, 143
eliminate_builtin, 144
eliminate_definition, 144
eliminate_definition_func, 144
eliminate_definition_pred, 144
eliminate_if, 144
eliminate_if_fmla, 144
eliminate_if_term, 144
eliminate_inductive, 144
eliminate_let, 144
eliminate_let_fmla, 144
eliminate_let_term, 144
eliminate_literal, 144
eliminate_mutual_recursion, 144
eliminate_recursion, 144
encoding_smt, 144
encoding_tptp, 144
exists, 144
hide, 144
hide_and_clear, 145
induction, 145
induction from, 145
induction_arg_pr, 145
induction_arg_ty_lex, 145
induction_pr, 146
induction_ty_lex, 146
inline_all, 146
inline_goal, 146
inline_trivial, 146
inst_rem, 147
instantiate, 146
introduce_premises, 147
intros, 147
intros_n, 147
inversion_arg_pr, 147
inversion_pr, 147
left, 147
pose, 148
remove, 148
replace, 148
revert, 148
rewrite, 149
rewrite with, 149
rewrite_list, 150
right, 150
simplify_array, 150
simplify_formula, 150
simplify_formula_and_task, 150
simplify_recursive_definition, 150
simplify_trivial_quantification, 150
simplify_trivial_quantification_in_goal, 151
split_all, 151
split_all_full, 151
split_goal, 151
split_goal_full, 153
split_intro, 153
split_premise, 153
split_premise_full, 153
subst, 153
unfold, 154
use_th, 154
tuples, 100
type invariant, 98

U

unfold
    transformation, 154
use_th
    transformation, 154

V

vc:annotation
    attribute, 156
vc:divergent
    attribute, 156
vc:keep_precondition
    attribute, 156
vc:sp
    attribute, 156
vc:white_box
    attribute, 156
vc:wp
    attribute, 156

W

wc
    command, 81
while loop, 95
why3 command line option
    -C, 61
    -L, 61
    --config, 61
    --debug, 62
    --debug-all, 62
    --extra-config, 61
    --help, 62
    --library, 61
    --list-debug-flags, 61
    --list-formats, 62
    --list-metas, 62
    --list-printers, 61
    --list-provers, 62

--list-transforms, 61
why3-doc command line option
--index, 78
--no-index, 78
--output, 78
--stdlib-url, 78
--title, 78
-o, 78
why3-execute command line option
--rac, 80
--rac-fail-cannot-reduce, 80
--rac-prover, 80
--rac-try-negate, 80
--use, 80
why3-extract command line option
-D, 81
--driver, 81
--flat, 81
--modular, 81
--recursive, 81
-o, 81
why3-pp command line option
--kind, 79
--output, 79
--prefix, 79
why3-prove command line option
-D, 64
-F, 64
-G, 64
-P, 64
-T, 64
--apply-transform, 64
--check-ce, 64
--driver, 64
--extra-expl-prefix, 64
--format, 64
--goal, 64
--prover, 64
--rac-prover, 64
--rac-try-negate, 64
--theory, 64
-a, 64
why3-replay command line option
--force, 73
--obsolete-only, 73
--prover, 73
--smoke-detector, 73
-q, 73
-s, 73
why3-session-html command line option
--add_pp, 77
--context, 77
--coqdoc, 77
--style, 77
-o, 77
why3-session-info command line option
--edited-files, 74
--print0, 74
--provers, 74
--stats, 74
why3-session-latex command line option
--longtable, 76
--style, 75
-e, 76
-o, 75
why3-session-update command line option
--rename-file, 78
WHY3CONFIG, 134