
The Why3 platform

Version 0.70, July 2011

François Bobot^{1,2}
Jean-Christophe Filliâtre^{1,2}
Claude Marché^{2,1}
Andrei Paskevich^{1,2}

¹ LRI, CNRS, Univ Paris-Sud, Orsay, F-91405

² INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893

©2010-2011 Univ Paris-Sud, CNRS, INRIA

This work has been partly supported by the ‘U3CAT’ national ANR project (ANR-08-SEGI-021-08, <http://frama-c.cea.fr/u3cat>) and by the Hi-Lite (<http://www.open-do.org/projects/hi-lite/>) FUI project of the System@tic competitiveness cluster.

Foreword

This is the manual for the Why platform version 3, or **Why3** for short. **Why3** is a complete reimplementaion [5] of the former Why platform [10] for program verification. Among the new features are: numerous extensions to the input language, a new architecture for calling external provers, and a well-designed API, allowing to use **Why3** as a software library. An important emphasis is put on modularity and genericity, giving the end user a possibility to easily reuse **Why3** formalizations or to add support for a new external prover if wanted.

Availability

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

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

See the file `INSTALL` for quick installation instructions, and Section 8.1 of this document for more detailed instructions.

Contact

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

Report any bug to the **Why3** Bug Tracking System: https://gforge.inria.fr/tracker/?atid=10293&group_id=2990&func=browse.

Acknowledgements

We gratefully thank the people who contributed to **Why3**, directly or indirectly: Romain Bardou, Simon Cruanes, Johannes Kanig, Stéphane Lescuyer, Simão Melo de Sousa, Guillaume Melquiond, Asma Tafat.

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 w.r.t. 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

Contents

Contents	5
I Tutorial	7
1 Getting Started	9
1.1 Hello Proofs	9
1.2 Getting Started with the GUI	9
1.3 Getting Started with the Why3 Command	15
2 The Why3 Language	17
3 The Why3ML Programming Language	25
3.1 Problem 1: Sum and Maximum	26
3.2 Problem 2: Inverting an Injection	28
3.3 Problem 3: Searching a Linked List	29
3.4 Problem 4: N-Queens	32
3.5 Problem 5: Amortized Queue	37
4 The Why3 API	41
4.1 Building Propositional Formulas	41
4.2 Building Tasks	42
4.3 Calling External Provers	43
4.4 Building Terms	45
4.5 Building Quantified Formulas	46
4.6 Building Theories	46
4.7 Applying transformations	46
4.8 Writing new functions on term	46
II Reference Manual	47
5 Language Reference	49
5.1 Lexical conventions	49
5.2 Why3 Language	51
5.3 Why3ML Language	56
6 Standard Library: Why3 Theories	59
6.1 Library <code>bool</code>	59
6.2 Library <code>int</code>	59

6.3	Library <code>real</code>	60
6.4	Library <code>floating_point</code>	60
6.5	Library <code>array</code>	60
6.6	Library <code>option</code>	61
6.7	Library <code>list</code>	61
7	Standard Library: Why3ML Modules	63
7.1	Library <code>ref</code>	63
7.2	Library <code>array</code>	63
7.3	Library <code>queue</code>	63
7.4	Library <code>stack</code>	63
7.5	Library <code>hashtbl</code>	63
7.6	Library <code>string</code>	64
8	Reference manuals for the Why3 tools	65
8.1	Compilation, Installation	65
8.2	Installation of external provers	66
8.3	The <code>why3config</code> command-line tool	66
8.4	The <code>why3</code> command-line tool	67
8.5	The <code>why3ide</code> GUI	68
8.6	The <code>why3ml</code> tool	71
8.7	The <code>why3bench</code> tool	71
8.8	The <code>why3replayer</code> tool	72
8.9	The <code>why3.conf</code> configuration file	73
8.10	Drivers of External Provers	74
8.11	Transformations	74
	Bibliography	77
	List of Figures	79
	Index	81

Part I

Tutorial

Chapter 1

Getting Started

1.1 Hello Proofs

The first and basic step in using Why3 is to write a suitable input file. When one wants to learn a programming language, you start by writing a basic program. Here we start by writing a file containing a basic set of goals.

Here is our first Why3 file, which is the file `examples/hello_proof.why` of the distribution.

```
theory HelloProof "My very first Why3 theory"

  goal G1 : true

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

  use import int.Int

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

end
```

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

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

1.2 Getting Started with the GUI

The graphical interface allows to browse into a file or a set of files, and check the validity of goals with external provers, in a friendly way. This section presents the basic use of this GUI. Please refer to Section 8.5 for a more complete description.

The GUI is launched on the file above as follows.

```
why3ide hello_proof.why
```

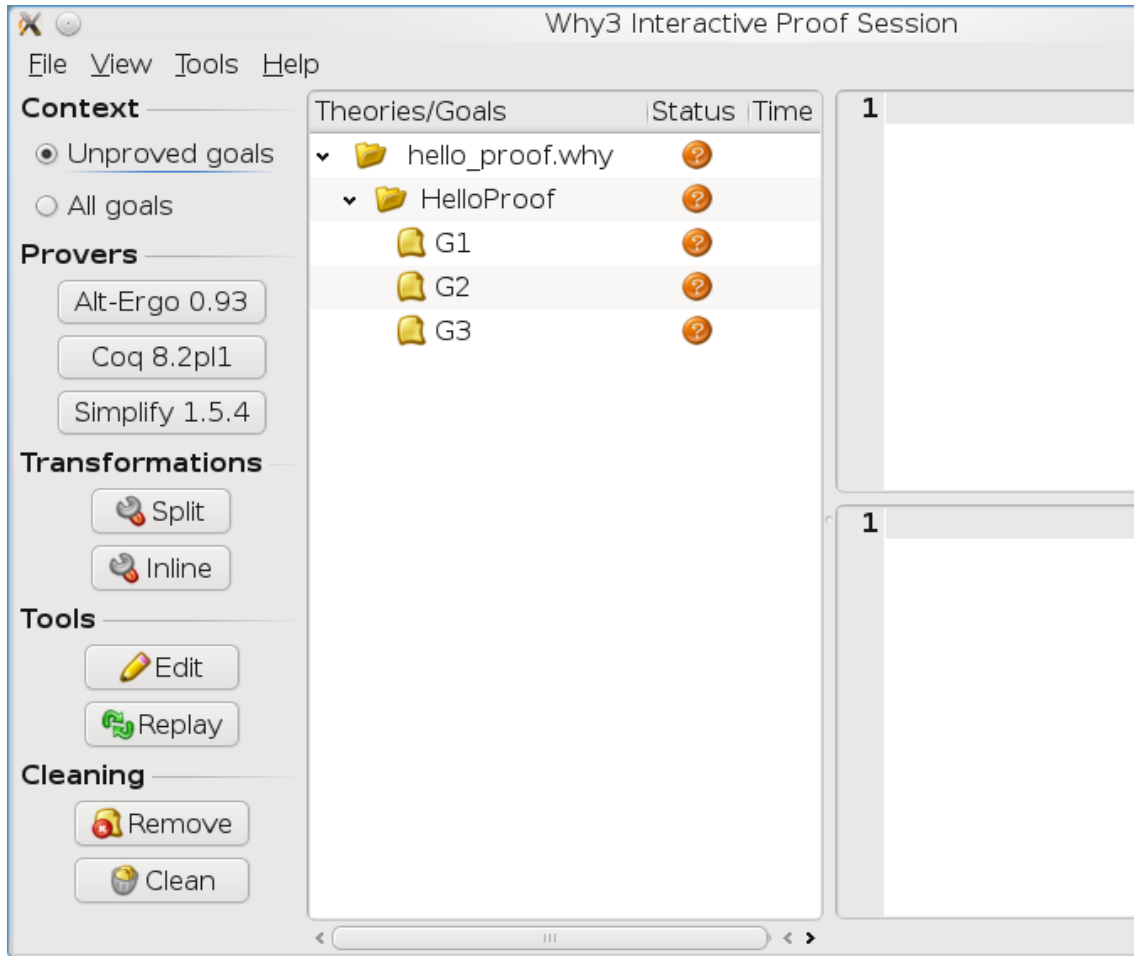


Figure 1.1: The GUI when started the very first time

When the GUI is started for the first time, you should get a window which looks like the screenshot of Figure 1.1.

The left column is a tool bar which provides different actions to apply on goals. The section “Provers” displays the provers that were detected as installed on your computer¹. Three provers were detected, in this case these are Alt-Ergo [6], Coq [3] and Simplify [8].

The middle part is a tree view that allows to browse inside the theories. In this tree view, we have a structured view of the file: this file contains one theory, itself containing three goals.

In Figure 1.2, we clicked on the row corresponding to goal G_1 . The *task* associated with this goal is then displayed on the top right, and the corresponding part of the input file is shown on the bottom right part.

Calling provers on goals

You are now ready to call these provers on the goals. Whenever you click on a prover button, this prover is called on the goal selected in the tree view. You can select several goals at a time, either by using multi-selection (typically by clicking while pressing the Shift or Ctrl key) or by selecting the parent theory or the parent file. Let us now select the

¹If not done yet, you must perform prover autodetection using `why3config -detect-provers`

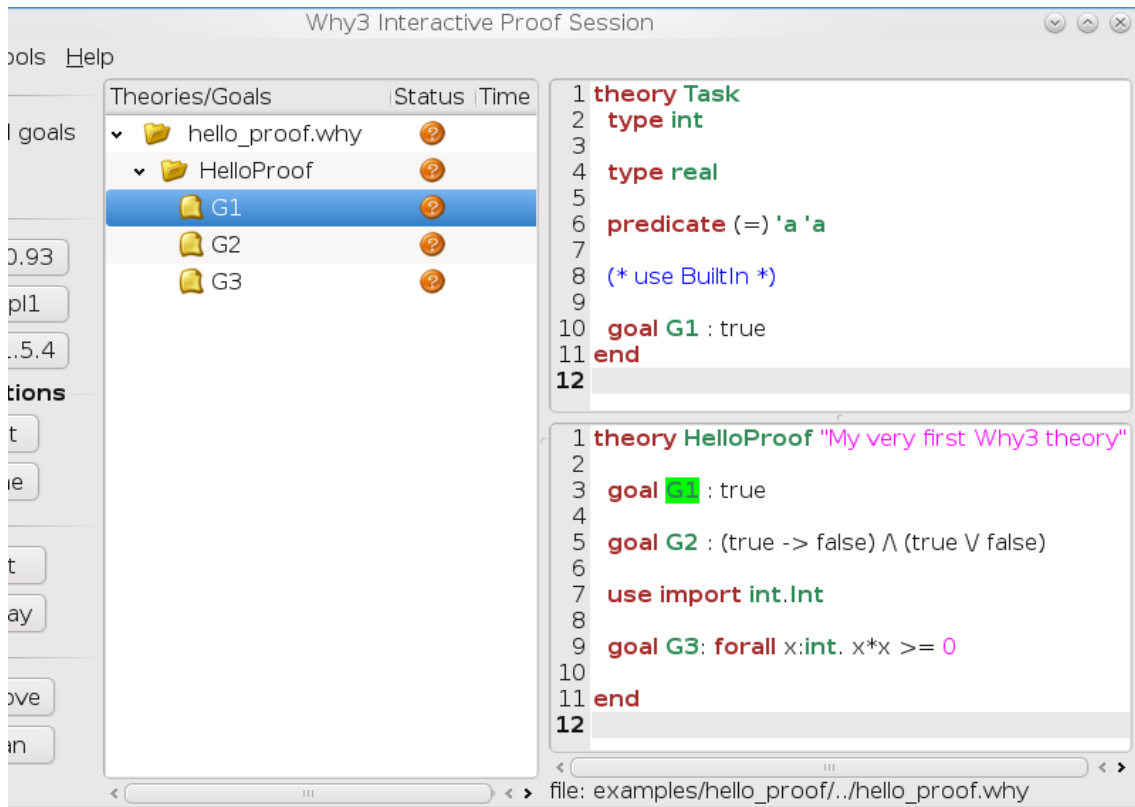


Figure 1.2: The GUI with goal G1 selected

theory “HelloProof” and click on the **Simplify** button. After a short time, you should get the display of Figure 1.3.

The goal G_1 is now marked with a green “checked” icon in the status column. This means that the goal is proved by the Simplify prover. On the contrary, the two other goals are not proved, they remain marked with an orange question mark.

You can immediately attempt to prove the remaining goals using another prover, *e.g.* Alt-Ergo, by clicking on the corresponding button. The goal G_3 should be proved now, but not G_2 .

Applying transformations

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

The first part of goal G_2 is still unproved. As a last resort, we can try to call the Coq proof assistant. The first step is to click on the **Coq** button. A new sub-row appear for Coq, and unsurprisingly the goal is not proved by Coq either. What can be done now is editing the proof: select that row and then click on the **Edit** button in section “Tools” of the toolbar. This should launch the Coq proof editor, which is `coqide` by default (see Section 8.5 for details on how to configure this). You get now a regular Coq file to fill in, as

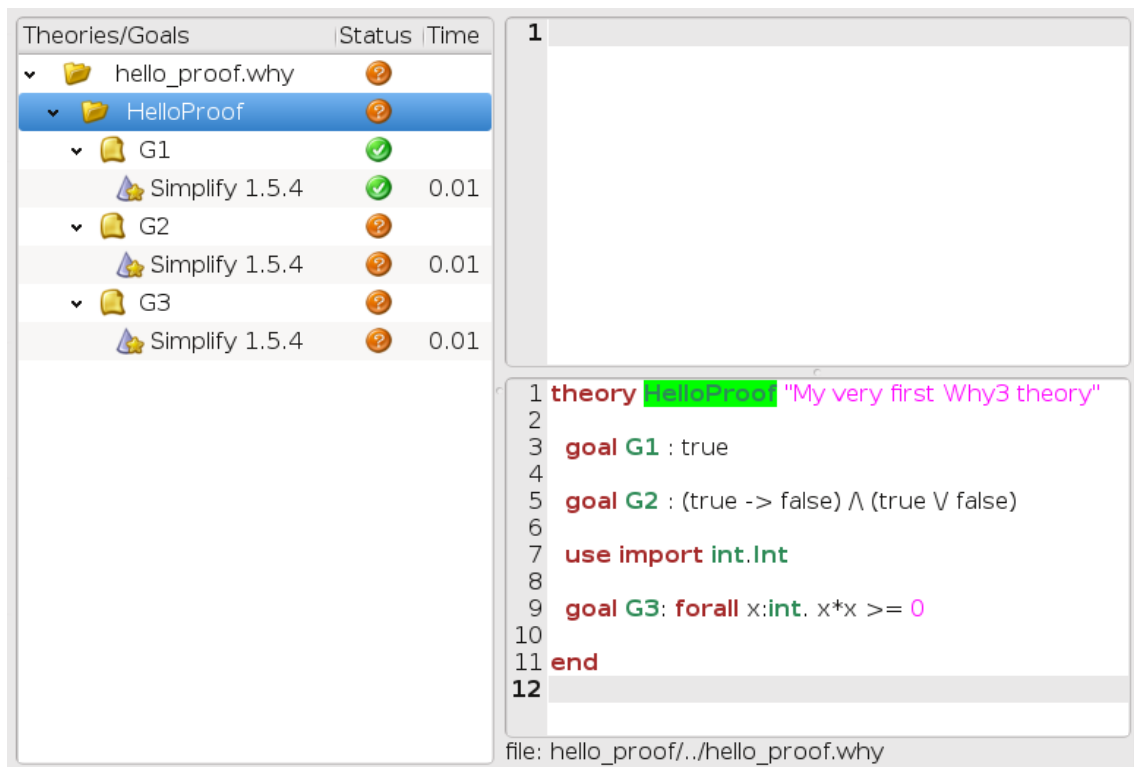
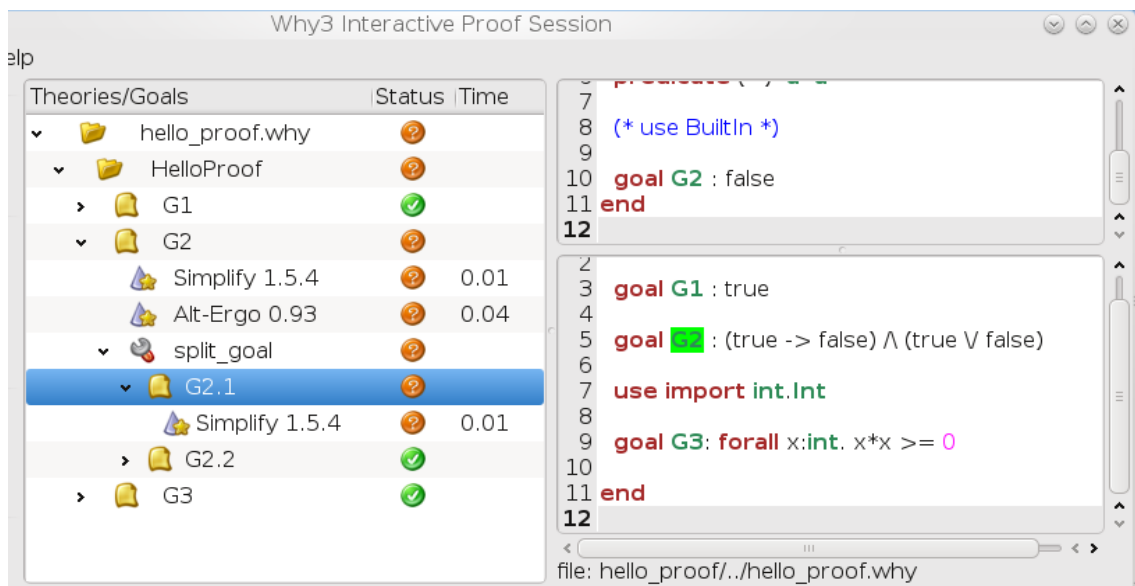
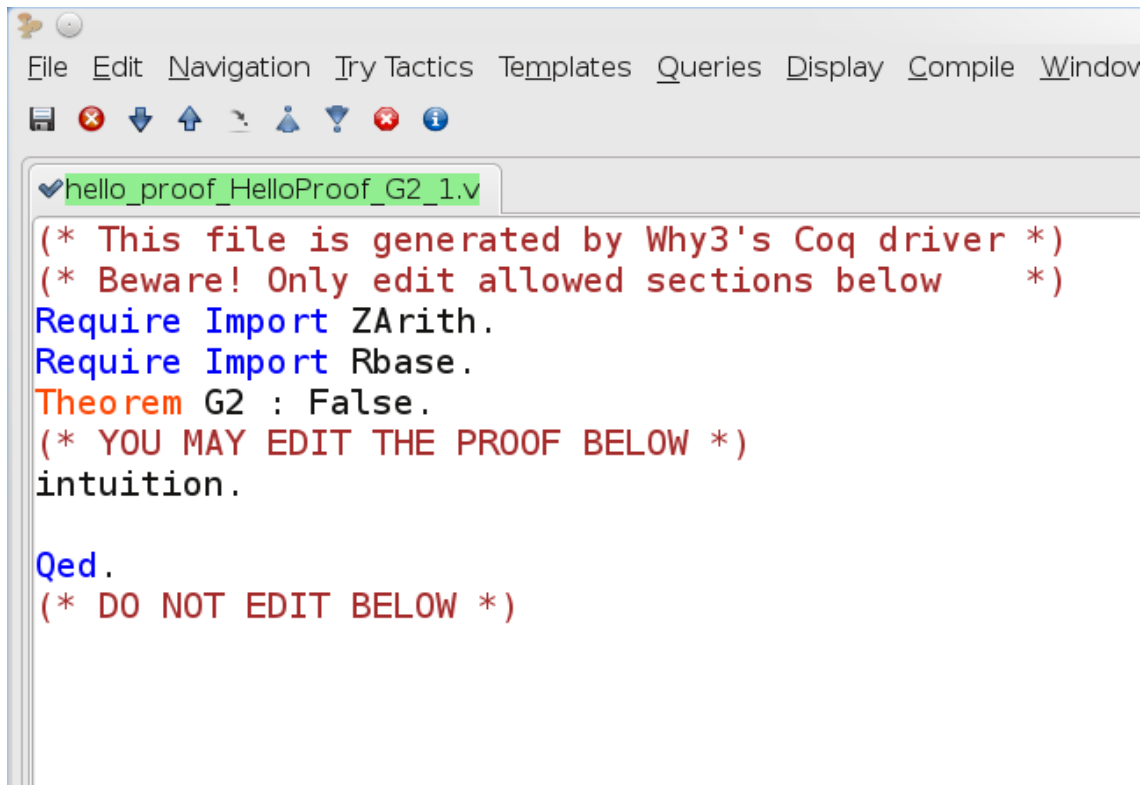


Figure 1.3: The GUI after Simplify prover is run on each goal

Figure 1.4: The GUI after splitting goal G_2 and collapsing proved goals

Figure 1.5: CoqIDE on subgoal 1 of G_2

shown on Figure 1.5. Please take care of the comments of this file. Only the part between the two last comments can be modified. Moreover, these comments themselves should not be modified at all, they are used to mark the part you modify, in order to regenerate the file if the goal is changed.

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.

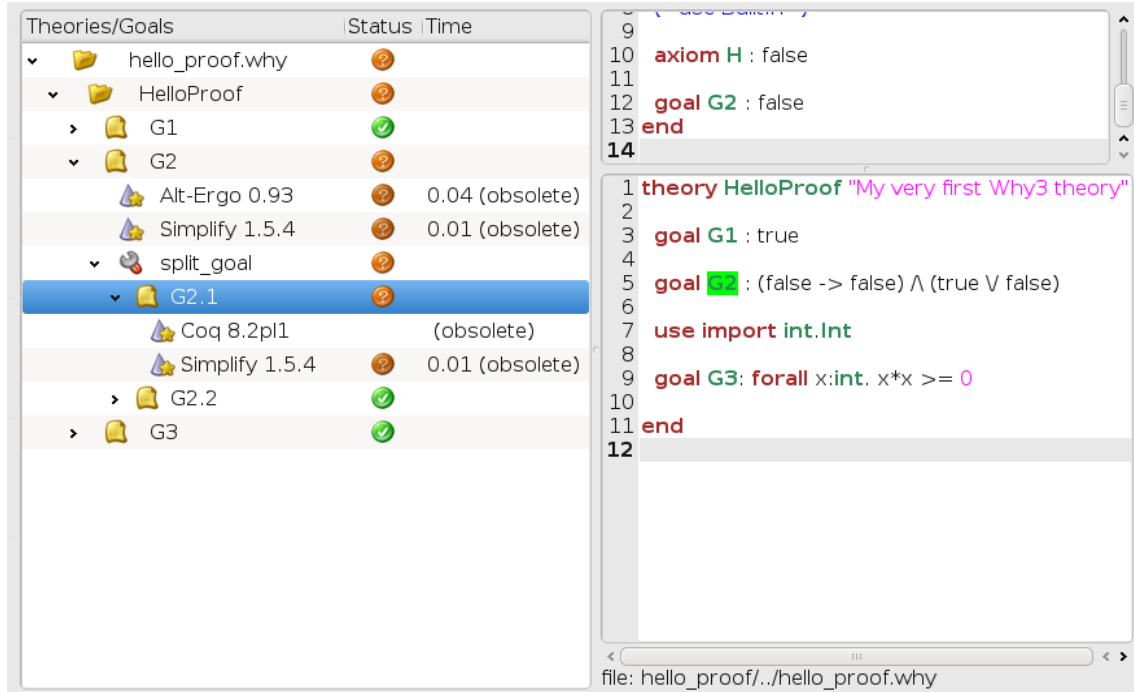
Modifying the input

Currently, the GUI does not allow to modify the input file. You must edit the file external by some editor of your choice. Let's assume we change the goal G_2 by replacing the first occurrence of true by false, *e.g.*

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

We can reload the modified file in the IDE using menu File/Reload, or the shortcut “Ctrl-R”. We get the tree view shown on Figure 1.6.

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

Figure 1.6: File reloaded after modifying goal G_2

Replaying obsolete proofs

Instead of pushing a prover's button to rerun its proofs, you can *replay* the existing but obsolete proof attempts, by clicking on the **Replay** button. By default, **Replay** only replays proofs that were successful before. If you want to replay all of them, you must select the context **all goals** at the top of the left tool bar.

Notice that replaying can be done in batch mode, using the `why3replayer` tool (see Section 8.8). For example, running the replayer on the `hello_proof` example is as follows (assuming G_2 still is $(\text{true} \rightarrow \text{false}) / (\text{true} \ \text{false})$).

```
$ why3replayer hello_proof
Info: found directory 'hello_proof' for the project
Opening session...[Xml warning] prolog ignored
[Reload] file '../hello_proof.why'
[Reload] theory 'HelloProof'
[Reload] transformation split_goal for goal G2
done
Progress: 9/9
2/3
  +--file ../hello_proof.why: 2/3
    +--theory HelloProof: 2/3
      +--goal G2 not proved
Everything OK.
```

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

Cleaning

You may want to clean some 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 clicking on button **Remove**. You must confirm the removal. Beware that there is no way to undo such a removal.

The **Clean** button performs an automatic removal of all proofs attempts that are unsuccessful, while there exists a successful proof attempt for the same goal.

1.3 Getting Started with the Why3 Command

The `why3` command allows to check the validity of goals with external provers, in batch mode. This section presents the basic use of this tool. Refer to Section 8.4 for a more complete description of this tool and all its command-line options.

The very first time you want to use Why, you should proceed with autodetection of external provers. We have already seen how to do it in the Why3 GUI. On the command line, this is done as follows (here “>” is the prompt):

```
> why3config --detect
```

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

```
> why3 --list-provers
Known provers:
  alt-ergo (Alt-Ergo)
  coq (Coq)
  simplify (Simplify)
```

The first word of each line is a unique identifier for the associated prover. We thus have now the three provers Alt-Ergo [6], Coq [3] and Simplify [8].

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

```
> why3 -P simplify hello_proof.why
hello_proof.why HelloProof G1 : Valid (0.10s)
hello_proof.why HelloProof G2 : Unknown: Unknown (0.01s)
hello_proof.why HelloProof G3 : Unknown: Unknown (0.00s)
```

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

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

```
> why3 -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  
split_intro
```

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

```
> why3 -P simplify hello_proof.why -a split_goal -T HelloProof -G G2
hello_proof.why HelloProof G2 : Unknown: Unknown (0.00s)
hello_proof.why HelloProof G2 : Valid (0.00s)
```

Section [8.11](#) gives the description of the various transformations available.

Chapter 2

The Why3 Language

This chapter describes the input syntax, and informally gives its semantics, illustrated by examples.

A Why3 text contains a list of *theories*. A theory is a list of *declarations*. Declarations introduce new types, functions and predicates, state axioms, lemmas and goals. These declarations can be directly written in the theory or taken from existing theories. The base logic of Why3 is first-order logic with polymorphic types.

Example 1: lists

The Figure 2.1 contains an example of Why3 input text, containing three theories. The first theory, **List**, declares a new algebraic type for polymorphic lists, `list 'a`. As in ML, `'a` stands for a type variable. The type `list 'a` has two constructors, `Nil` and `Cons`. Both constructors can be used as usual function symbols, respectively of type `list 'a` and `'a × list 'a → list 'a`. We deliberately make this theory that short, for reasons which will be discussed later.

The next theory, **Length**, introduces the notion of list length. The `use import List` command indicates that this new theory may refer to symbols from theory **List**. These symbols are accessible in a qualified form, such as `List.list` or `List.Cons`. The `import` qualifier additionally allows us to use them without qualification.

Similarly, the next command `use import int.Int` adds to our context the theory `int.Int` from the standard library. The prefix `int` indicates the file in the standard library containing theory `Int`. Theories referred to without prefix either appear earlier in the current file, *e.g.* **List**, or are predefined.

The next declaration defines a recursive function, `length`, which computes the length of a list. The `function` and `predicate` keywords are used to introduce function and predicate symbols, respectively. Why3 checks every recursive, or mutually recursive, definition for termination. Basically, we require a lexicographic and structural descent for every recursive call for some reordering of arguments. Notice that matching must be exhaustive and that every `match` expression must be terminated by the `end` keyword.

Despite using higher-order “curried” syntax, Why3 does not permit partial application: function and predicate arities must be respected.

The last declaration in theory **Length** is a lemma stating that the length of a list is non-negative.

The third theory, **Sorted**, demonstrates the definition of an inductive predicate. Every such definition is a list of clauses: universally closed implications where the consequent is an instance of the defined predicate. Moreover, the defined predicate may only occur in positive positions in the antecedent. For example, a clause:

```

theory List
  type list 'a = Nil | Cons 'a (list 'a)
end

theory Length
  use import List
  use import int.Int

  function length (l : list 'a) : int =
    match l with
    | Nil      -> 0
    | Cons _ r -> 1 + length r
    end

  lemma Length_nonnegative : forall l:list 'a. length l >= 0
end

theory Sorted
  use import List
  use import int.Int

  inductive sorted (list int) =
    | Sorted_Nil :
      sorted Nil
    | Sorted_One :
      forall x:int. sorted (Cons x Nil)
    | Sorted_Two :
      forall x y : int, l : list int.
      x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
end

```

Figure 2.1: Example of Why3 text.

```

| Sorted_Bad :
  forall x y : int, l : list int.
  (sorted (Cons y l) -> y > x) -> sorted (Cons x (Cons y l))

```

would not be allowed. This positivity condition assures the logical soundness of an inductive definition.

Note that the type signature of `sorted` predicate does not include the name of a parameter (see `l` in the definition of `length`): it is unused and therefore optional.

Example 1 (continued): lists and abstract orderings

In the previous section we have seen how a theory can reuse the declarations of another theory, coming either from the same input text or from the library. Another way to referring to a theory is by “cloning”. A `clone` declaration constructs a local copy of the cloned theory, possibly instantiating some of its abstract (*i.e.* declared but not defined) symbols.

```

theory Order
  type t
  predicate (<=) t t

  axiom Le_refl : forall x : t. x <= x
  axiom Le_asym : forall x y : t. x <= y -> y <= x -> x = y
  axiom Le_trans: forall x y z : t. x <= y -> y <= z -> x <= z
end

theory SortedGen
  use import List
  clone import Order as O

  inductive sorted (l : list t) =
    | Sorted_Nil :
      sorted Nil
    | Sorted_One :
      forall x:t. sorted (Cons x Nil)
    | Sorted_Two :
      forall x y : t, l : list t.
      x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
  end

theory SortedIntList
  use import int.Int
  clone SortedGen with type O.t = int, predicate O.<= = (<=)
end

```

Figure 2.2: Example of Why3 text (continued).

Consider the continued example in Figure 2.2. We write an abstract theory of partial orders, declaring an abstract type `t` and an abstract binary predicate `<=`. Notice that an infix operation must be enclosed in parentheses when used outside a term. We also specify three axioms of a partial order.

There is little value in `use`'ing such a theory: this would constrain us to stay with the type `t`. However, we can construct an instance of theory `Order` for any suitable type and predicate. Moreover, we can build some further abstract theories using order, and then instantiate those theories.

Consider theory `SortedGen`. In the beginning, we `use` the earlier theory `List`. Then we make a simple `clone` theory `Order`. This is pretty much equivalent to copy-pasting every declaration from `Order` to `SortedGen`; the only difference is that Why3 traces the history of cloning and transformations and drivers often make use of it (see Section 8.10).

Notice an important difference between `use` and `clone`. If we `use` a theory, say `List`, twice (directly or indirectly: *e.g.* by making `use` of both `Length` and `Sorted`), there is no duplication: there is still only one type of lists and a unique pair of constructors. On the contrary, when we `clone` a theory, we create a local copy of every cloned declaration, and the newly created symbols, despite having the same names, are different from their originals.

Returning to the example, we finish theory `SortedGen` with a familiar definition of predicate `sorted`; this time we use the abstract order on the values of type `t`.

Now, we can instantiate theory `SortedGen` to any ordered type, without having to retype the definition of `sorted`. For example, theory `SortedIntList` makes clone of `SortedGen` (*i.e.* copies its declarations) substituting type `int` for type `0.t` of `SortedGen` and the default order on integers for predicate `0.(<=)`. Why3 will control that the result of cloning is well-typed.

Several remarks ought to be made here. First of all, why should we clone theory `Order` in `SortedGen` if we make no instantiation? Couldn't we write `use import Order as 0` instead? The answer is no, we could not. When cloning a theory, we only can instantiate the symbols declared locally in this theory, not the symbols imported with `use`. Therefore, we create a local copy of `Order` in `SortedGen` to be able to instantiate `t` and `(<=)` later.

Secondly, when we instantiate an abstract symbol, its declaration is not copied from the theory being cloned. Thus, we will not create a second declaration of type `int` in `SortedIntList`.

The mechanism of cloning bears some resemblance to modules and functors of ML-like languages. Unlike those languages, Why3 makes no distinction between modules and module signatures, modules and functors. Any Why3 theory can be `use`'d directly or instantiated in any of its abstract symbols.

The command-line tool `why3` (described in Section 1.3), allows us to see the effect of cloning. If the input file containing our example is called `lists.why`, we can launch the following command:

```
> why3 lists.why -T SortedIntList
```

to see the resulting theory `SortedIntList`:

```
theory SortedIntList
  (* use BuiltIn *)
  (* use Int *)
  (* use List *)

  axiom Le_refl : forall x:int. x <= x
  axiom Le_asym : forall x:int, y:int. x <= y -> y <= x -> x = y
  axiom Le_trans : forall x:int, y:int, z:int. x <= y -> y <= z
    -> x <= z

  (* clone Order with type t = int, predicate (<=) = (<=),
     prop Le_trans1 = Le_trans, prop Le_asym1 = Le_asym,
     prop Le_refl1 = Le_refl *)

  inductive sorted (list int) =
    | Sorted_Nil : sorted (Nil:list int)
    | Sorted_One : forall x:int. sorted (Cons x (Nil:list int))
    | Sorted_Two : forall x:int, y:int, l:list int. x <= y ->
      sorted (Cons y l) -> sorted (Cons x (Cons y l))

  (* clone SortedGen with type t1 = int, predicate sorted1 = sorted,
     predicate (<=) = (<=), prop Sorted_Two1 = Sorted_Two,
     prop Sorted_One1 = Sorted_One, prop Sorted_Nil1 = Sorted_Nil,
     prop Le_trans2 = Le_trans, prop Le_asym2 = Le_asym,
```

```

    prop Le_refl2 = Le_refl *)
end

```

In conclusion, let us briefly explain the concept of namespaces in **Why3**. Both **use** and **clone** instructions can be used in three forms (the examples below are given for **use**, the semantics for **clone** is the same):

- **use List as L** — every symbol *s* of theory **List** is accessible under the name **L.s**. The **as L** part is optional, if it is omitted, the name of the symbol is **List.s**.
- **use import List as L** — every symbol *s* from **List** is accessible under the name **L.s**. It is also accessible simply as *s*, but only up to the end of the current namespace, *e.g.* the current theory. If the current theory, that is the one making **use**, is later used under the name **T**, the name of the symbol would be **T.L.s**. (This is why we could refer directly to the symbols of **Order** in theory **SortedGen**, but had to qualify them with **0.** in **SortedIntList**.) As in the previous case, **as L** part is optional.
- **use export List** — every symbol *s* from **List** is accessible simply as *s*. If the current theory is later used under the name **T**, the name of the symbol would be **T.s**.

Why3 allows to open new namespaces explicitly in the text. In particular, the instruction “**clone import Order as 0**” can be equivalently written as:

```

namespace import 0
  clone export Order
end

```

However, since **Why3** favours short theories over long and complex ones, this feature is rarely used.

Example 2: Einstein’s problem

We now consider another, slightly more complex example: how to use **Why3** to solve a little puzzle known as “Einstein’s logic problem”¹. The problem is stated as follows. Five persons, of five different nationalities, live in five houses in a row, all painted with different colors. These five persons own different pets, drink different beverages and smoke different brands of cigars. We are given the following information:

- The Englishman lives in a red house;
- The Swede has dogs;
- The Dane drinks tea;
- The green house is on the left of the white one;
- The green house’s owner drinks coffee;
- The person who smokes Pall Mall has birds;
- The yellow house’s owner smokes Dunhill;
- In the house in the center lives someone who drinks milk;

¹This **Why3** example was contributed by Stéphane Lescuyer.

- 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 "Einstein's problem"
```

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

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

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

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

It introduces two functions, namely *Color.of* and *Color.to*, from houses to colors and colors to houses, respectively, and the two 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 "Hints"
  use import Einstein

```

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

```

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

```

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

```

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

```

Finally, we declare the goal in the fourth theory:

```

theory Problem "Goal of Einstein's problem"
  use import Einstein
  use import EinsteinHints

  goal G: Pet.to Fish = German
end

```

and we are ready to use `Why3` to discharge this goal with any prover of our choice.

Chapter 3

The Why3ML Programming Language

This chapter describes the Why3ML programming language. A Why3ML input text contains a list of theories (see chapter 2) and/or modules. Modules extend theories with *programs*. Programs can use all types, symbols, and constructs from the logic. They also provide extra features:

- In a record type declaration, some fields can be declared `mutable`.
- There are programming constructs with no counterpart in the logic:
 - mutable field assignment;
 - sequence;
 - loops;
 - exceptions;
 - local and anonymous functions;
 - annotations: pre- and postconditions, assertions, loop invariants.
- A program function can be non-terminating or can be proved to be terminating using a variant (a term together with a well-founded order relation).
- An abstract program type t can be introduced with a logical *model* τ : inside programs, t is abstract, and inside annotations, t is an alias for τ .

Programs are contained in files with suffix `.mlw`. They are handled by the tool `why3ml`, which has a command line similar to `why3`. For instance

```
% why3ml myfile.mlw
```

will display the verification conditions extracted from modules in file `myfile.mlw`, as a set of corresponding theories, and

```
% why3ml -P alt-ergo myfile.mlw
```

will run the SMT solver Alt-Ergo on these verification conditions. Program files are also handled by the GUI tool `why3ide`. See Chapter 8 for more details regarding command lines.

As an introduction to Why3ML, we use the five problems from the VSTTE 2010 verification competition [15]. The source code for all these examples is contained in Why3's distribution, in sub-directory `examples/programs/`.

3.1 Problem 1: Sum and Maximum

The first problem is stated as follows:

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

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

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

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

```
module MaxAndSum
```

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

```
use import int.Int
```

We are also going to use references and arrays from Why3ML's standard library, so we import the corresponding modules, with a similar declaration:

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

The additional keyword `module` means that we are looking for `.mlw` files from the standard library (namely `ref.mlw` and `array.mlw` here), instead of `.why` files. Modules `Ref` and `Array` respectively provide a type `ref 'a` for references and a type `array 'a` for arrays (see Chapter 7), together with useful operations and traditional syntax.

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) = ...
```

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

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

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
```

```

module MaxAndSum

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

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

end

```

Figure 3.1: Solution for VSTTE'10 competition problem 1.

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 to prove `sum <= n * max`. The loop invariant is introduced with the keyword `invariant`, immediately after the keyword `do`.

```

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

```

There is no need to introduce a variant, as the termination of a `for` loop is automatically guaranteed. This completes module `MaxAndSum`. Figure 3.1 shows the whole code. We can now proceed to its verification. Running `why3ml`, or better `why3ide`, on file `max_sum.mlw` will show a single verification condition with name `WP_parameter_max_sum`. Discharging this verification condition with an automated theorem prover will not succeed, most likely, as it involves non-linear arithmetic. Repeated applications of goal splitting and calls to

SMT solvers (within `why3ide`) will typically leave a single, unsolved goal, which reduces to proving the following sequent:

$$s \leq i \times \max, \max < a[i] \vdash s + a[i] \leq (i + 1) \times a[i].$$

This is easily discharged using an interactive proof assistant such as Coq, and thus completes the verification.

3.2 Problem 2: Inverting an Injection

The second problem is stated as follows:

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

We may assume that A is surjective and we have to prove that the resulting array is also injective. The code is immediate, since it is as simple as

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

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

```
module InvertingAnInjection
  use import int.Int
  use import module 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: int. 0 <= i < n -> 0 <= j < n -> i <> j -> a[i] <> a[j]

predicate surjective (a: array int) (n: int) =
  forall i: int. 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: int. 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:

```

module InvertingAnInjection

  use import int.Int
  use import module array.Array

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

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

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

end

```

Figure 3.2: Solution for VSTTE'10 competition problem 2.

```

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

```

Here we chose to have array `b` as argument; returning a freshly allocated array would be equally simple. The whole module is given Figure 3.2. The verification conditions for function `inverting` are easily discharged automatically, thanks to the lemma.

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

```
module SearchingALinkedList
  use import int.Int
  use 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:int. 0 <= j < i -> nth j l <> Some 0
```

and another for a non-successful search,

```
predicate no_zero (l: list int) =
  forall j:int. 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) = match l with
  | Nil      -> i
  | Cons x r -> if x = 0 then i else search (i+1) r
end
```

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

We first prove the termination of this recursive function. It amounts to give it a *variant*, that is an term integer term which stays non-negative and strictly decreases at each recursive call. Here it is as simple as the length of l :

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

(It is worth pointing out that variants are not limited to natural numbers. Any other type equipped with a well-founded order relation can be used instead.) 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
```

```

module SearchingALinkedList

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

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

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

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

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

end

```

Figure 3.3: Solution for VSTTE'10 competition problem 3.

Solving the problem is simply a matter of calling `search` with 0 as first argument. The code is given Figure 3.3. The verification conditions are all discharged automatically.

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

```

use import module ref.Ref
use import list.HdTl

```

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

```

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

```

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) =
  { l <> Nil }
  match l with Nil -> absurd | Cons _ t -> t end
  { tl l = Some result }
```

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 =
  { }
  let i = ref 0 in
  let s = ref l in
  while !s <> Nil && head !s <> 0 do
    invariant { ... }
    variant { length !s }
    i := !i + 1;
    s := tail !s
  done;
  !i
  { ... same postcondition as search_list ... }
```

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.

3.4 Problem 4: N-Queens

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

```
module NQueens
  use import int.Int
  use import module 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 not 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:int. 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) =
  { 0 <= pos < length board }
  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
{ result=True <-> is_consistent board pos }

```

The assertion in the exception handler is a cut for SMT solvers. This first part of the solution is given Figure 3.4.

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

```

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

predicate solution (board: array int) (pos: int) =
  is_board board pos /\

```

```

module NQueens
  use import int.Int
  use import module 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:int. 0 <= q < pos -> consistent_row board pos q

  exception Inconsistent int

  let check_is_consistent (board: array int) (pos: int) =
    { 0 <= pos < length board }
    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
    { result=True <-> is_consistent board pos }

```

Figure 3.4: Solution for VSTTE'10 competition problem 4 (1/2).

```
forall q:int. 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:int. 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) variant {n-pos} =
```

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

```
{ length board = n /\ 0 <= pos <= n /\ solution board pos }
'Init:
```

We place a code mark `'Init` immediately after the precondition to be able to refer to the value of `board` in the pre-state. 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 (at board 'Init) pos /\
  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. 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`.

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

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) =
  { 0 <= length board = n }
  bt_queens board n 0
  { forall b:array int. length b = n -> is_board b n -> not (solution b n) }
  | Solution -> { solution board n }
```

This second part of the solution is given Figure 3.5. With the help of a few auxiliary lemmas — not given here but available from Why3's sources — the verification conditions are all discharged automatically, including the verification of the lemmas themselves.

```

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

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

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

exception Solution

let rec bt_queens (board: array int) (n: int) (pos: int) variant { n - pos } =
  { length board = n /\ 0 <= pos <= n /\ solution board pos }
  'Init:
  if pos = n then raise Solution;
  for i = 0 to n - 1 do
    invariant {
      eq_board board (at board 'Init) pos /\
      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
  { (* no solution *)
    eq_board board (old board) pos /\
    forall b:array int. length b = n -> is_board b n ->
      eq_board board b pos -> not (solution b n) }
  | Solution ->
  { (* a solution *)
    solution board n }

let queens (board: array int) (n: int) =
  { 0 <= length board = n }
  bt_queens board n 0
  { forall b:array int. length b = n -> is_board b n -> not (solution b n) }
  | Solution -> { solution board n }
end

```

Figure 3.5: Solution for VSTTE'10 competition problem 4 (2/2).

3.5 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* [12] 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 which imports all list operations we will require: length, reversal, and concatenation.

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

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

```
type queue 'a = {| front: list 'a; lenf: int;
                  rear : list 'a; lenr: int; |}
```

We start with the definition of the data type invariant, as a predicate `inv`. It makes use of the ability to chain several equalities and inequalities.

```
predicate inv (q: queue 'a) =
  length q.front = q.lenf >= length q.rear = q.lenr
```

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 reversed rear list.

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

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

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

The postcondition is twofold: the returned queue satisfies its invariant and represents the empty sequence. 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) =
  { inv q /\ sequence q <> Nil }
  match q.front with
  | Nil      -> absurd
  | Cons x _ -> x
  end
  { hd (sequence q) = Some result }

```

Note the presence of the invariant in the precondition, which is required to prove the absurdity of the first branch (if `q.front` is `Nil`, then so should be `sequence q`).

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`, it is a good idea to introduce a smart constructor `create` which 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) =
  { lf = length f /\ lr = length 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 | }
  { inv result /\ sequence result = f ++ reverse r }

```

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) =
  { inv q /\ sequence q <> Nil }
  match q.front with
  | Nil      -> absurd
  | Cons _ r -> create r (q.lenf - 1) q.rear q.lenr
  end
  { inv result /\ tl (sequence q) = Some (sequence result) }

```

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) =
  { inv q }
  create q.front q.lenf (Cons x q.rear) (q.lenr + 1)
  { inv result /\ sequence result = sequence q ++ Cons x Nil }

```

The code is given Figure 3.6. The verification conditions are all discharged automatically.

```

module AmortizedQueue
  use import int.Int
  use export list.ListRich

  type queue 'a = {| front: list 'a; lenf: int;
                    rear : list 'a; lenr: int; |}

  predicate inv (q: queue 'a) =
    length q.front = q.lenf >= length q.rear = q.lenr

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

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

  let head (q: queue 'a) =
    { inv q /\ sequence q <> Nil }
    match q.front with
    | Nil      -> absurd
    | Cons x _ -> x
    end
    { hd (sequence q) = Some result }

  let create (f: list 'a) (lf: int) (r: list 'a) (lr: int) =
    { lf = length f /\ lr = length 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 |}
    { inv result /\ sequence result = f ++ reverse r }

  let tail (q: queue 'a) =
    { inv q /\ sequence q <> Nil }
    match q.front with
    | Nil      -> absurd
    | Cons _ r -> create r (q.lenf - 1) q.rear q.lenr
    end
    { inv result /\ tl (sequence q) = Some (sequence result) }

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

```

Figure 3.6: Solution for VSTTE'10 competition problem 5.

Chapter 4

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/>.

We assume the reader has a fair knowledge of the OCaml language. Notice that the Why3 library must be installed, see Section 8.1. The OCaml code given below is available in the source distribution as [examples/use_api.ml](#).

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 $true \vee false$.

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

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

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

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

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

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

```
ocamlc str.cma unix.cma nums.cma dynlink.cma \
-I +ocamlgraph -I +why3 graph.cma why.cma f.ml -o f
```

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

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

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

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

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

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

As expected, the output is as follows.

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

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

4.2 Building Tasks

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

```
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, however, 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 logic declarations in the task.

```

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

```

The argument `None` is the declarations of logic symbols means that they do not have any definition.

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 Why configuration file `why3.conf`, as it was built using the `why3config` command line tool or the **Detect Provers** menu of the graphical IDE. The following API calls allow to access the content of this configuration file.

```

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

```

The type `'a Util.Mstr.t` is a map indexed by strings. This map can provide the set of existing provers. In the following, we directly attempt to access the prover `Alt-Ergo`, which is known to be identified with id `"alt-ergo"`.

```

(* the [prover alt-ergo] section of the config file *)
let alt_ergo : Whyconf.config_prover =
  try
    Util.Mstr.find "alt-ergo" provers
  with Not_found ->
    eprintf "Prover alt-ergo not installed or not configured@";
    exit 0

```

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

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

```
(* calls Alt-Ergo *)
let result1 : Call_provers.prover_result =
  Call_provers.wait_on_call
    (Driver.prove_task ~command:alt_ergo.Whyconf.command
      alt_ergo_driver task1 ()) ()
(* prints Alt-Ergo answer *)
let () = printf "@[0n task 1, alt-ergo answers %a@]@."
  Call_provers.print_prover_result result1
```

This way to call a prover is in general too naive, since it may never return if the prover runs without time limit. The function `prove_task` has two optional parameters: `timelimit` is the maximum allowed running time in seconds, and `memlimit` is the maximum allowed memory in megabytes. The type `prover_result` is a record with three fields:

- `pr_answer`: the prover answer, explained below;
- `pr_output`: the output of the prover, i.e. both standard output and the standard error of the process (a redirection in `why3.conf` is required);
- `pr_time`: the time taken by the prover, in seconds.

A `pr_answer` is a sum of several kind of answers:

- **Valid**: the task is valid according to the prover.
- **Invalid**: the task is invalid.
- **Timeout**: the prover exceeds the time or memory limit.
- **Unknown *msg***: the prover can't determine if the task is valid; the string parameter *msg* indicates some extra information.
- **Failure *msg***: the prover reports a failure, i.e. 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 (i.e. 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
      ~timelimit: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` (see Chap 6).

```
let two : Term.term = Term.t_const (Term.ConstInt "2")
let four : Term.term = Term.t_const (Term.ConstInt "4")
let int_theory : Theory.theory =
  Env.find_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 is 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 $\forall x : \text{int}. x * x \geq 0$. The first step is to obtain the symbols from `Int`.

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

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

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

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

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

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

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

4.6 Building Theories

[TO BE COMPLETED]

4.7 Applying transformations

[TO BE COMPLETED]

4.8 Writing new functions on term

[TO BE COMPLETED]

Part II

Reference Manual

Chapter 5

Language Reference

This chapter gives the grammar and semantics for Why3 and Why3ML input files.

5.1 Lexical conventions

Lexical conventions are common to Why3 and Why3ML.

Comments. Comments are enclosed by `(*` and `*)` and can be nested.

Strings. Strings are enclosed in double quotes `"`. Double quotes can be inserted in strings using the backslash character `\`. In the following, strings are referred to with the non-terminal *string*.

Identifiers. The syntax distinguishes lowercase and uppercase identifiers and, similarly, lowercase and uppercase qualified identifiers.

<i>lalpha</i>	::=	<code>a - z</code> <code>_</code>
<i>ualpha</i>	::=	<code>A - Z</code>
<i>alpha</i>	::=	<i>lalpha</i> <i>ualpha</i>
<i>lident</i>	::=	<i>lalpha</i> (<i>alpha</i> <i>digit</i> <code>'</code>)*
<i>uident</i>	::=	<i>ualpha</i> (<i>alpha</i> <i>digit</i> <code>'</code>)*
<i>ident</i>	::=	<i>lident</i> <i>uident</i>
<i>lqualid</i>	::=	<i>lident</i> <i>uqualid</i> . <i>lident</i>
<i>uqualid</i>	::=	<i>uident</i> <i>uqualid</i> . <i>uident</i>

Constants. The syntax for constants is given in Figure 5.1. Integer and real constants have arbitrary precision. Integer constants may be given in base 16, 10, 8 or 2. Real constants may be given in base 16 or 10.

Operators. Prefix and infix operators are built from characters organized in four categories (*op-char-1* to *op-char-4*).

<i>digit</i>	::=	0 - 9	
<i>hex-digit</i>	::=	<i>digit</i> a - f A - F	
<i>oct-digit</i>	::=	0 - 7	
<i>bin-digit</i>	::=	0 1	
<i>integer</i>	::=	<i>digit</i> (<i>digit</i> <i>_</i>)*	decimal
		(0x 0X) <i>hex-digit</i> (<i>hex-digit</i> <i>_</i>)*	hexadecimal
		(0o 0O) <i>oct-digit</i> (<i>oct-digit</i> <i>_</i>)*	octal
		(0b 0B) <i>bin-digit</i> (<i>bin-digit</i> <i>_</i>)*	binary
<i>real</i>	::=	<i>digit</i> ⁺ <i>exponent</i>	decimal
		<i>digit</i> ⁺ . <i>digit</i> [*] <i>exponent</i> [?]	
		<i>digit</i> [*] . <i>digit</i> ⁺ <i>exponent</i> [?]	
		(0x 0X) <i>hex-real</i> <i>h-exponent</i>	hexadecimal
<i>hex-real</i>	::=	<i>hex-digit</i> ⁺	
		<i>hex-digit</i> ⁺ . <i>hex-digit</i> [*]	
		<i>hex-digit</i> [*] . <i>hex-digit</i> ⁺	
<i>exponent</i>	::=	(e E) (- +) [?] <i>digit</i> ⁺	
<i>h-exponent</i>	::=	(p P) (- +) [?] <i>digit</i> ⁺	

Figure 5.1: Syntax for constants.

<i>op-char-1</i>	::=	= < > ~
<i>op-char-2</i>	::=	+ -
<i>op-char-3</i>	::=	* / %
<i>op-char-4</i>	::=	! \$ & ? @ ^ . : #
<i>op-char</i>	::=	<i>op-char-1</i> <i>op-char-2</i> <i>op-char-3</i> <i>op-char-4</i>
<i>infix-op-1</i>	::=	<i>op-char</i> [*] <i>op-char-1</i> <i>op-char</i> [*]
<i>infix-op</i>	::=	<i>op-char</i> ⁺
<i>prefix-op</i>	::=	<i>op-char</i> ⁺
<i>bang-op</i>	::=	! <i>op-char-4</i> [*] ? <i>op-char-4</i> [*]

Infix operators are classified into 4 categories, according to the characters they are built from:

- level 4: operators containing only characters from *op-char-4*;
- level 3: those containing characters from *op-char-3* or *op-char-4*;
- level 2: those containing characters from *op-char-2*, *op-char-3* or *op-char-4*;
- level 1: all other operators (non-terminal *infix-op-1*).

Labels. Identifiers, terms, formulas, program expressions can all be labeled, either with a string, or with a location tag.

<i>label</i>	<code>::=</code>	<i>string</i>				
			<code>#</code>	<i>filename</i>	<i>digit</i> ⁺	<i>digit</i> ⁺ <i>digit</i> ⁺ <code>#</code>
<i>filename</i>	<code>::=</code>	<i>string</i>				

A location tag consists of a file name, a line number, and starting and ending characters.

5.2 Why3 Language

Terms. The syntax for terms is given in Figure 5.2. The various constructs have the following priorities and associativities, from lowest to greatest priority:

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

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

Type Expressions. The syntax for type expressions is the following:

<i>type</i>	<code>::=</code>	<i>lqualid</i>	<i>type</i> [*]	type symbol
			<code>'</code> <i>lident</i>	type variable
			<code>()</code>	empty tuple type
			<code>(type (, type)⁺)</code>	tuple type
			<code>(type)</code>	parentheses

Built-in types are `int`, `real`, and tuple types. Note that the syntax for type expressions notably differs from the usual ML syntax (*e.g.* the type of polymorphic lists is written `list 'a`, not `'a list`).

Formulas. The syntax for formulas is given Figure 5.3. The various constructs have the following priorities and associativities, from lowest to greatest priority:

<i>term</i>	<i>::=</i>	<i>integer</i>	integer constant
		<i>real</i>	real constant
		<i>lqualid</i>	symbol
		<i>prefix-op term</i>	
		<i>bang-op term</i>	
		<i>term infix-op term</i>	
		<i>term [term]</i>	brackets
		<i>term [term <- term]</i>	ternary brackets
		<i>lqualid term⁺</i>	function application
		<i>if formula then term</i>	
		<i>else term</i>	conditional
		<i>let pattern = term in term</i>	local binding
		<i>match term (, term)* with</i>	
		<i>(term-case)⁺ end</i>	pattern matching
		<i>(term (, term)⁺)</i>	tuple
		<i>{ field-value⁺ }</i>	record
		<i>term . lqualid</i>	field access
		<i>{ term with field-value⁺ }</i>	field update
		<i>term : type</i>	cast
		<i>label term</i>	label
		<i>' uident</i>	code mark
		<i>(term)</i>	parentheses
<i>term-case</i>	<i>::=</i>	<i>pattern -> term</i>	
<i>pattern</i>	<i>::=</i>	<i>pattern pattern</i>	or pattern
		<i>pattern , pattern</i>	tuple
		<i>-</i>	catch-all
		<i>lident</i>	variable
		<i>uident pattern*</i>	constructor
		<i>(pattern)</i>	parentheses
		<i>pattern as lident</i>	binding
<i>field-value</i>	<i>::=</i>	<i>lqualid = term ;</i>	

Figure 5.2: Syntax for terms.

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

Figure 5.3: Syntax for formulas.

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

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

Notice that there are two symbols for the conjunction: `and` and `&&`, and similarly for disjunction. There are logically equivalent, but may be treated slightly differently by some

transformation, *e.g.* the `split` transformation transforms A and B into subgoals A and B , whereas it transforms $A \ \&\& \ B$ into subgoals A and $A \rightarrow B$.

Theories. The syntax for theories is given Figure 5.4.

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

$file \quad ::= \quad theory^*$

<i>theory</i>	<code>::=</code>	<code>theory uident label* decl* end</code>	
<i>decl</i>	<code>::=</code>	<code>type type-decl (with type-decl)*</code> <code> function function-decl (with logic-decl)*</code> <code> predicate predicate-decl (with logic-decl)*</code> <code> inductive inductive-decl (with inductive-decl)*</code> <code> axiom ident : formula</code> <code> lemma ident : formula</code> <code> goal ident : formula</code> <code> use imp-exp tqualid (as uident-opt)?</code> <code> clone imp-exp tqualid (as uident-opt)? subst?</code> <code> namespace import? uident-opt decl* end</code>	
<i>type-decl</i>	<code>::=</code>	<code>lident label* (' lident label*) type-defn</code>	
<i>type-defn</i>	<code>::=</code>	<code> = type</code> <code> = ? type-case (type-case)*</code> <code> = { record-field (; record-field)* }</code>	abstract type alias type algebraic type record type
<i>type-case</i>	<code>::=</code>	<code>uident label* type-param*</code>	
<i>record-field</i>	<code>::=</code>	<code>lident label* : type</code>	
<i>logic-decl</i>	<code>::=</code>	<code>function-decl</code> <code> predicate-decl</code>	
<i>function-decl</i>	<code>::=</code>	<code>lident label* type-param* : type</code> <code> lident label* type-param* : type = term</code>	
<i>predicate-decl</i>	<code>::=</code>	<code>lident label* type-param*</code> <code> lident label* type-param* = formula</code>	
<i>type-param</i>	<code>::=</code>	<code>' lident</code> <code> lqualid</code> <code> (lident⁺ : type)</code> <code> (type (, type)*)</code> <code> ()</code>	
<i>inductive-decl</i>	<code>::=</code>	<code>lident label* type-param* =</code> <code> ? ind-case (ind-case)*</code>	
<i>ind-case</i>	<code>::=</code>	<code>ident label* : formula</code>	
<i>imp-exp</i>	<code>::=</code>	<code>(import export)?</code>	
<i>uident-opt</i>	<code>::=</code>	<code>uident _</code>	
<i>subst</i>	<code>::=</code>	<code>with (, subst-elt)⁺</code>	
<i>subst-elt</i>	<code>::=</code>	<code>type lqualid = lqualid</code> <code> function lqualid = lqualid</code> <code> predicate lqualid = lqualid</code> <code> namespace (uqualid .) = (uqualid .)</code> <code> lemma uqualid</code> <code> goal uqualid</code>	
<i>tqualid</i>	<code>::=</code>	<code>uident ident (. ident)[*] . uident</code>	

Figure 5.4: Syntax for theories.

$type-v$	$::=$	$type \mid (type-v)$	parentheses
		$ type-v \rightarrow type-c$	
		$ type-v-binder \rightarrow type-c$	
$type-v-binder$	$::=$	$lident \ label^* : type-v$	
$type-v-param$	$::=$	$(type-v-binder)$	
$type-c$	$::=$	$type-v$	
		$ pre \ type-v \ effect \ post$	
$effect$	$::=$	$reads^? \ writes^? \ raises^?$	
$reads$	$::=$	$reads \ tr-term^+$	
$writes$	$::=$	$writes \ tr-term^+$	
$raises$	$::=$	$raises \ uqualid^+$	
pre	$::=$	$annotation$	
$post$	$::=$	$annotation \ post-exn^*$	
$post-exn$	$::=$	$ uqualid \rightarrow annotation$	
$annotation$	$::=$	$\{\} \mid \{ formula \}$	

Figure 5.5: Syntax for program types.

5.3 Why3ML Language

Types. The syntax for program types is given in figure 5.5.

Expressions. The syntax for program expressions is given in figure 5.6.

Modules. The syntax for modules is given in figure 5.7. Any declaration which is accepted in a theory is also accepted in a module. Additionally, modules can introduce record types with mutable fields and declarations which are specific to programs (global variables, functions, exceptions).

Files. A Why3ML input file is a (possibly empty) list of theories and modules.

$file$	$::=$	$(theory \mid module)^*$
--------	-------	--------------------------

A theory defined in a Why3ML file can only be used within that file. If a theory is supposed to be reused from other files, be they Why3 or Why3ML files, it should be defined in a Why3 file.

<i>expr</i>	::=	<i>integer</i>	integer constant
		<i>real</i>	real constant
		<i>lqualid</i>	symbol
		<i>prefix-op</i> <i>expr</i>	
		<i>expr</i> <i>infix-op</i> <i>expr</i>	
		<i>expr</i> [<i>expr</i>]	brackets
		<i>expr</i> [<i>expr</i>] <- <i>expr</i>	brackets assignment
		<i>expr</i> [<i>expr</i> <i>infix-op-l</i> <i>expr</i>]	ternary brackets
		<i>expr</i> <i>expr</i> ⁺	function application
		fun <i>type-v-param</i> ⁺ -> <i>triple</i>	lambda abstraction
		let rec <i>recfun</i> (with <i>recfun</i>) [*]	recursive functions
		if <i>expr</i> then <i>expr</i> (else <i>expr</i>) [?]	conditional
		<i>expr</i> ; <i>expr</i>	sequence
		loop <i>loop-annot</i> end	infinite loop
		while <i>expr</i> do <i>loop-annot</i> <i>expr</i> done	while loop
		for <i>lident</i> = <i>expr</i> to-downto <i>expr</i>	for loop
		do <i>loop-inv</i> <i>expr</i> done	
		assert <i>annotation</i>	assertion
		absurd	
		raise <i>uqualid</i>	exception raising
		raise (<i>uqualid</i> <i>expr</i>)	
		try <i>expr</i> with (<i>handler</i>) ⁺ end	exception catching
		any <i>type-c</i>	
		let <i>pattern</i> = <i>expr</i> in <i>expr</i>	local binding
		match <i>expr</i> (, <i>expr</i>) [*] with	pattern matching
		(<i>expr-case</i>) ⁺ end	
		(<i>expr</i> (, <i>expr</i>) ⁺)	tuple
		{ <i>field-value</i> ⁺ }	record
		<i>exopr</i> . <i>lqualid</i>	field access
		<i>expr</i> . <i>lqualid</i> <- <i>expr</i>	field assignment
		{ <i>expr</i> with <i>field-value</i> ⁺ }	field update
		<i>expr</i> : <i>type</i>	cast
		label <i>expr</i>	label
		' <i>uident</i> : <i>expr</i>	code mark
		(<i>expr</i>)	parentheses
<i>expr-case</i>	::=	<i>pattern</i> -> <i>expr</i>	
<i>field-value</i>	::=	<i>lqualid</i> = <i>expr</i> ;	
<i>triple</i>	::=	<i>expr</i>	
		<i>pre</i> <i>expr</i> <i>post</i>	Hoare triple
<i>assert</i>	::=	assert assume check	
<i>to-downto</i>	::=	to downto	
<i>loop-annot</i>	::=	<i>loop-inv</i> [?] <i>variant</i> [?]	
<i>loop-inv</i>	::=	invariant <i>annotation</i>	
<i>variant</i>	::=	variant { <i>term</i> } (with <i>lqualid</i>) [?]	

Figure 5.6: Syntax for program expressions.

<i>module</i>	<code>::=</code>	<code>module uident label* mdecl* end</code>	
<i>mdecl</i>	<code>::=</code>	<code>decl</code>	theory declaration
		<code>type mtype-decl (with mtype-decl)*</code>	mutable types
		<code>let lident label* pgm-defn</code>	
		<code>let rec recfun (with recfun)*</code>	
		<code>val lident label* pgm-decl</code>	
		<code>exception lident label* type?</code>	
		<code>use imp-exp module tqualid (as uident-opt)?</code>	
		<code>namespace import? uident-opt mdecl* end</code>	
<i>mtype-decl</i>	<code>::=</code>	<code>lident label* (‘ lident label*)* mtype-defn</code>	
<i>mtype-defn</i>	<code>::=</code>		abstract type
		<code>= type</code>	alias type
		<code>= ? type-case (type-case)*</code>	algebraic type
		<code>= { mrecord-field (; mrecord-field)* }</code>	record type
<i>mrecord-field</i>	<code>::=</code>	<code>mutable? lident label* : type</code>	
<i>pgm-decl</i>	<code>::=</code>	<code>: type-v</code>	
		<code>type-v-param⁺ : type-c</code>	
<i>pgm-defn</i>	<code>::=</code>	<code>type-v-param⁺ (: type)? = triple</code>	
		<code>= fun type-v-param⁺ -> triple</code>	

Figure 5.7: Syntax for modules.

Chapter 6

Standard Library: Why3 Theories

We provide here a short description of logic symbols defined in the standard library. Only the most general-purpose ones are described. For more details, one should directly read the corresponding file, or alternatively, use the `why3` with option `-T` and a qualified theory name, for example:

```
> why3 -T bool.Ite
theory Ite
  (* use BuiltIn *)

  (* use Bool *)

  function ite (b:bool) (x:'a) (y:'a) : 'a =
    match b with
    | True -> x
    | False -> y
    end
end
```

In the following, for each library, we describe the (main) symbols defined in it.

6.1 Library bool

Bool boolean data type `bool` with constructors `True` and `False`; operations `andb`, `orb`, `xorb`, `notb`.

Ite polymorphic if-then-else operator written as `ite`.

6.2 Library int

Int basic operations `+`, `-` and `*`; comparison operators `<`, `>`, `>=` and `<=`.

Abs absolute value written as `abs`.

EuclideanDivision division and modulo, where division rounds down, written as `div` and `mod`.

ComputerDivision division and modulo, where division rounds to zero, also written as `div` and `mod`.

MinMax `min` and `max` operators.

6.3 Library `real`

Real basic operations `+`, `-`, `*` and `/`; comparison operators.

RealInfix basic operations with alternative syntax `+. , -. , *. , /. , <. , >. , <=. , >=. ,` to allow simultaneous use of integer and real operators.

Abs absolute value written as `abs`.

MinMax `min` and `max` operators.

FromInt operator `from_int` to convert an integer to a real.

Truncate conversion operators from real to integers: `truncate` rounds to 0, `floor` rounds down and `ceil` rounds up.

Square operators `sqr` and `sqrt` for square and square root.

ExpLog functions `exp`, `log`, `log2`, and `log10`.

Power function `pow` with two real parameters.

Trigonometry functions `cos`, `sin`, `tan`, and `atan`. Constant `pi`.

Hyperbolic functions `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`.

Polar functions `hypot` and `atan2`.

6.4 Library `floating_point`

This library provides a theory of IEEE-754 floating-point numbers. It is inspired by [1].

Rounding type `mode` with 5 constants `NearestTiesToEven`, `ToZero`, `Up`, `Down` and `NearTiesToAway`.

SpecialValues handling of infinities and NaN.

GenFloat generic floats parameterized by the maximal representable number. Functions `round`, `value`, `exact`, `model`, predicate `no_overflow`.

Single instance of `GenFloat` for 32-bits single precision numbers.

Double instance of `GenFloat` for 64-bits double precision numbers.

6.5 Library `array`

Array polymorphic arrays, a.k.a maps. Type `t` parameterized by both the type of indices and the type of data. Functions `get` and `set` to access and update arrays. Function `create_const` to produce an array initialized by a given constant.

ArrayLength arrays indexed by integers and holding their length. Function `length`.

ArrayRich additional functions on arrays indexed by integers. Functions `sub` and `app` to extract a sub-array and append arrays.

6.6 Library option

Option data type `option 'a` with constructors `None` and `Some`.

6.7 Library list

List data type `list 'a` with constructors `Nil` and `Cons`.

Length function `length`

Mem function `mem` for testing for list membership.

Nth function `nth` for extract the n -th element.

HdTl functions `hd` and `tl`.

Append function `append`, concatenation of lists.

Reverse function `reverse` for list reversal.

Sorted predicate `sorted` for lists of integers.

NumOcc number of occurrences in a list.

Permut list permutations.

Induction structural induction on lists.

Map list map operator.

Chapter 7

Standard Library: Why3ML Modules

7.1 Library ref

Ref references *i.e.* mutable variables: type `ref 'a` and functions `ref` for creation, `(!)` for access, and `(:=)` for mutation

Refint references with additional functions `incr` and `decr` over integer references

7.2 Library array

Array polymorphic arrays (type `array 'a`, infix syntax `a[i]` for access and `a[i] ← e` for update, functions `length`, `make`, `append`, `sub`, `copy`, `fill`, and `blit`)

ArraySorted an array of integers is sorted (`array_sorted_sub` and `array_sorted`)

ArrayEq two arrays are identical (`array_eq_sub` and `array_eq`)

ArrayPermut two arrays are permutation of each other (`permut_sub` and `permut`)

7.3 Library queue

Queue polymorphic mutable queues (type `t 'a` and functions `create`, `push`, `pop`, `top`, `clear`, `copy`, `is_empty`, `length`)

7.4 Library stack

Stack polymorphic mutable stacks (type `t 'a` and functions `create`, `push`, `pop`, `top`, `clear`, `copy`, `is_empty`, `length`)

7.5 Library hashtable

Hashtbl hash tables with monomorphic keys (type `key`) and polymorphic values (type `t 'a` of hash tables, syntax `h[k]` for access, functions `create`, `clear`, `add`, `mem`, `find`, `find_all`, `copy`, `remove`, and `replace`)

7.6 Library string

Char

String

Chapter 8

Reference manuals for the Why3 tools

8.1 Compilation, Installation

Compilation of Why3 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, version 3.10 or higher. 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 tools, additional OCaml libraries are needed:

- For the IDE: the Lablgtk2 library for 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>

- For `why3bench`: The OCaml bindings of the sqlite3 library. For Debian-based Linux distributions, you can install the package

```
libsqlite3-ocaml-dev
```

It is also installable from sources, available from the site http://ocaml.info/home/ocaml_sources.html#ocaml-sqlite3

When configuration is finished, you can compile Why3.

```
make
```

Local use, without installation

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

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

The Why3 executables are then available in the subdirectory `bin/`.

Installation of the Why3 library

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

```
make byte opt
make install_lib
```

8.2 Installation of 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 these provers before compiling and installing Why.

For installation of external provers, please look at the Why provers tips page <http://why.lri.fr/provers.en.html>.

For configuring Why3 to use the provers, follow instructions given in Section 8.3.

8.3 The why3config command-line tool

Why3 must be configured to access external provers. Typically, this is done by running either the command line tool

```
why3config
```

or using the menu

```
File/Detect provers
```

of the IDE. This must be redone each time a new prover is installed.

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

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

The provers which are typically looked for are

- Alt-Ergo [4, 6]: <http://alt-ergo.lri.fr>

- CVC3 [2]: <http://cs.nyu.edu/acsys/cvc3/>
- Coq [3]: <http://coq.inria.fr>
- Eprover [14]: <http://www4.informatik.tu-muenchen.de/~schulz/WORK/eprover.html>
- Gappa [11]: <http://gappa.gforge.inria.fr/>
- Simplify [8]: <http://secure.ucd.ie/products/opensource/Simplify/>
- Spass: <http://www.spass-prover.org/>
- Vampire: <http://www.voronkov.com/vampire.cgi>
- VeriT: <http://www.verit-solver.org/>
- Yices [9]: <http://yices.csl.sri.com/>, only versions 1.xx since versions 2.xx do not support quantifiers
- Z3 [7]: <http://research.microsoft.com/en-us/um/redmond/projects/z3/>

`why3config` also detects the plugins installed in the Why3 plugins directory (*e.g.* `/usr/local/lib/why3/plugins`). A plugin must register itself as a parser, a transformation or a printer, as explained in the corresponding section.

If the user's configuration file is already present, `why3config` will only reset unset variables to default value, but will not try to detect provers. The option `-detect-provers` should be used to force Why3 to detect again the available provers and to replace them in the configuration file. The option `-detect-plugins` will do the same for plugins.

8.4 The why3 command-line tool

Why3 is primarily used to call provers on goals contained in an input file. By default, such a file must be written in Why3 language and have the extension `.why`. However, a dynamically loaded plugin can register a parser for some other format of logical problems, *e.g.* TPTP or SMTlib.

The `why3` tool 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 8.9.
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. The `-list-format` option gives the list of registered parsers.
5. Extract the selected goals inside each of the selected theories into tasks. The goals and theories are selected using the options `-G/-goal` and `-T/-theory`. The option `-T/-theory` applies to the last file appearing on the command line, the option `-G/-goal` applies to the last 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 transformation requested with `-a/-apply-transform` in their order of appearance on the command line. `-list-transforms` list the known transformations, plugins can add more of them.
7. Apply the driver selected with the `-D/-driver` option, or the driver of the prover selected with `-P/-prover` option. `-list-provers` lists the known provers, i.e. the ones which appear in the configuration file.
8. If the option `-P/-prover` is given, call the selected prover on each generated task and print the results. If the option `-D/-driver` is given, print each generated task using the format specified in the selected driver.

The provers can answer the following output:

Valid the goal is proved in the given context,

Unknown the prover stop by itself to search,

Timeout the prover doesn't have enough time,

Failure an error occurred,

Invalid the prover know the goal can't be proved

The option `-bisect` change the behaviors of why3. With this option, `-P/-prover` and `-o/-output` must be given and a valid goal must be selected. The last step executed by why3 is replaced by computing a minimal set (in the great majority of the case) of declarations which still prove the goal. Currently it doesn't use any information provided by the prover, it call the prover multiple time with reduced context. The minimal set of declarations is then written in the prover syntax into a file located in the directory given to the `-o/-output` option.

8.5 The why3ide GUI

The basic usage of the GUI is described by the tutorial of Section 1.2. We describe here the command-line options and the actions of the various menus and buttons of the interface.

Command-line options

-I *d*: adds *d* in the load path, to search for theories.

Left toolbar actions

Context The context in which the other tools below will apply. If “only unproved goals” is selected, no action will ever be applied to an already proved goal. If “all goals”, then actions are performed even if the goal is already proved. The second choice allows to compare provers on the same goal.

Provers To each detected prover corresponds to a button in this prover framed box. Clicking on this button starts the prover on the selected goal(s).

Split This splits the current goal into subgoals if it is a conjunction of two or more goals.

Inline If the goal is headed by a defined predicate symbol, expands it with this definition.

Edit Start an editor on the selected task.

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

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

Replay Replay all obsolete proofs

If “only unproved goals” is selected, only formerly successful proofs are rerun. If “all goals”, then all obsolete proofs are rerun, successful or not.

Remove Removes a proof attempt or a transformation.

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

Menus

Menu File

Add File adds a file in the GUI

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

Reload reloads the input files from disk, and update the session state accordingly

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.

Quit exits the GUI

Menu View

Expand All expands all the rows of the tree view

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

Menu Tools A copy of the tools already available in the left toolbar, plus:

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

Menu Help A very short online help, and some information about this software.

Preferences

The preferences window allows you customize

- the default editor to use when the **Edit** button is pressed. This might be overridden for a specific prover (the only way to do that for the moment is to manually edit the config file)
- the time limit given to provers, in seconds
- the maximal number of simultaneous provers allowed to run in parallel.

- The policy for saving session:
 - 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 save automatically, you must use menu **File/Save session** to save when wanted
 - ask whether to save: on exit, a popup window ask whether you want to save or not.

Structure of the database file

The 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 (file*)>
<!ATTLIST why3session name CDATA #REQUIRED>

<!ELEMENT file (theory*)>
<!ATTLIST file name CDATA #REQUIRED>
<!ATTLIST file verified CDATA #REQUIRED>
<!ATTLIST file expanded CDATA #IMPLIED>

<!ELEMENT theory (goal*)>
<!ATTLIST theory name CDATA #REQUIRED>
<!ATTLIST theory verified CDATA #REQUIRED>
<!ATTLIST theory expanded CDATA #IMPLIED>

<!ELEMENT goal (proof*, transf*)>
<!ATTLIST goal name CDATA #REQUIRED>
<!ATTLIST goal expl CDATA #IMPLIED>
<!ATTLIST goal proved CDATA #REQUIRED>
<!ATTLIST goal sum CDATA #REQUIRED>
<!ATTLIST goal expanded CDATA #IMPLIED>

<!ELEMENT proof (result|undone)>
<!ATTLIST proof prover CDATA #REQUIRED>
<!ATTLIST proof timelimit CDATA #REQUIRED>
<!ATTLIST proof edited CDATA #IMPLIED>
<!ATTLIST proof obsolete CDATA #IMPLIED>

<!ELEMENT result EMPTY>
<!ATTLIST result status (valid|invalid|unknown|timeout|failure) #REQUIRED>
<!ATTLIST result time CDATA #IMPLIED>

<!ELEMENT undone EMPTY>

<!ELEMENT transf (goal*)>
<!ATTLIST transf name CDATA #REQUIRED>

```

```
<!ATTLIST transf proved CDATA #REQUIRED>
<!ATTLIST transf expanded CDATA #IMPLIED>
```

8.6 The why3ml tool

`why3ml` is an additional layer on `Why3` library for generating verification conditions from `Why3ML` programs. The command-line of `why3ml` is identical to that of `why3`, but also accepts files with extension `.mlw` as input files containing `Why3ML` modules (see chapter 3 and section 5.3). Modules are turned into theories containing verification conditions as goals, and then `why3ml` behaves exactly as `why3` for the remaining of the process. Note that files with extension `.mlw` can also be loaded in `why3ide`.

For those who want to experiment with `Why3ML`, many examples are provided in `examples/programs`.

8.7 The why3bench tool

The `why3bench` tool adds a scheduler on top of the `Why3` library. `why3bench` is designed to compare various components of automatic proofs: automatic provers, transformations, definitions of a theory. For that goal it tries to prove predefined goals using each component to compare. `why3bench` allows to output the comparison in various formats:

- `csv`: the simpler and more informative format, the results are represented in an array, the rows corresponds to the compared components, the columns correspond to the result (Valid,Unknown,Timeout,Failure,Invalid) and the CPU time taken in seconds.
- `average`: summarizes the number of the five different answers for each component. It also gives the average time taken.
- `timeline`: for each component it gives the number of valid goals along the time (10 slices between 0 and the longest time a component takes to prove a goal)

The compared components can be defined in an *rc-file*, `examples/programs/prgbench.rc` is such an example. More generally a bench configuration file:

```
[probs "myprobs"]
  file = "examples/monbut.why" #relatives to the rc file
  file = "examples/monprogram.mlw"
  theory = "monprogram.T"
  goal = "monbut.T.G"

  transform = "split_goal" #applied in this order
  transform = "..."
  transform = "..."

[tools "mytools"]
  prover = cvc3
  prover = altergo
  #or only one
  driver = "..."
  command = "..."
```

```

loadpath = "..." #added to the one in why3.conf
loadpath = "..."

timelimit = 30
memlimit = 300

use = "toto.T" #use the theory toto (allow to add metas)

transform = "simplify_array" #only 1 to 1 transformation

[bench "mybench"]
  tools = "mytools"
  tools = ...
  probs = "myprobs"
  probs = ...
  timeline = "prgbench.time"
  average = "prgbench.avg"
  csv = "prgbench.csv"

```

Such a file can define three families **tools**, **probs**, **bench**. The sections **tools** define a set of components to compare, the sections **probs** define a set of goals on which to compare some components and the sections **bench** define which components to compare using which goals. It refers by name to the sections **tools** and **probs** defined in the same file. The order of the definitions is irrelevant. Notice that **loadpath** in a family **tools** can be used to compare different axiomatizations.

One can run all the bench given in one bench configuration file with **why3bench** :

```
why3bench -B path_to_my_bench.rc
```

8.8 The why3replayer tool

The purpose of the **why3replayer** tool is to execute the proofs stored in a Why3 session file, as the one produced by the IDE. Its main goal is to play non-regression tests, *e.g.* you can find in **examples/regtests.sh** a script that runs regression tests on all the examples.

The tool is invoked in a terminal or a script using

```
why3replayer [options] <project directory>
```

The session file **why3session.xml** stored in the given directory is loaded and all the proofs it contains are rerun. Then, any difference 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, independently of the fact the considered goal is proved or not. When all the proof runs are done, a summary of what is proved or not is displayed using a tree-shape pretty print, similar to the IDE tree view after doing “Collapse proved goals”, that is when a goal, a theory or a file is fully proved the subtree is not shown.

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.

- option `-s`: suppresses the output of the final tree view
- option `-I <path>`: suppresses the output of the final tree view

8.9 The *why3.conf* configuration file

One can use a custom configuration file. *why3config* and other *why3* tools use priorities for which user's configuration file to consider:

- the file specified by the `-C` or `--config` options,
- the file specified by the environment variable `WHY3CONFIG` if set.
- the file `$HOME/.why3.conf` (`$USERPROFILE/.why3.conf` under Windows) or, in the case of local installation, *why3.conf* in Why3 sources top directory.

If none of these files exists, a built-in default configuration is used.

The configuration file is a human-readable text file, which consists of association pairs arranged in sections. Here follows an example of configuration file.

```
[main ]
loadpath = "/usr/local/share/why3/theories"
magic = 2
memlimit = 0
running_provers_max = 2
timelimit = 10

[ide ]
default_editor = "emacs"
task_height = 384
tree_width = 438
verbose = 0
window_height = 779
window_width = 638

[prover coq]
command = "coqc %f"
driver = "/usr/local/share/why3/drivers/coq.drv"
editor = "coqide"
name = "Coq"
version = "8.2pl2"

[prover alt-ergo]
command = "why3-cpulimit %t %m alt-ergo %f"
driver = "/usr/local/share/why3/drivers/alt_ergo.drv"
editor = ""
name = "Alt-Ergo"
version = "0.91"
```

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 only one identifier, *main* and *ide* in the

example, or it can be composed by a family name and one family argument, **prover** is one family name, **coq** and **alt-ergo** are the family argument.

Inside a section, one key can be associated with an integer (eg -555), a boolean (true, false) or a string (e.g. "emacs"). One key can appear only once except if its a multi-value key. The order of apparition of the keys inside a section matter only for the multi-value key.

8.10 Drivers of External Provers

The drivers of external provers are readable files, in directory **drivers**. Experimented users can modify them to change the way the external provers are called, in particular which transformations are applied to goals.

[TO BE COMPLETED LATER]

8.11 Transformations

Here is a quick documentation of provided transformations. We give first the non-splitting ones, e.g. those which produce one goal as result, and others which produces any number of goals.

Notice that the set of available transformations in your own installation is given by

```
why3 --list-transforms
```

Non-splitting transformations

eliminate__algebraic Replaces algebraic data types by first-order definitions [13]

eliminate__builtin Suppress definitions of symbols which are declared as builtin in the driver, i.e. with a “syntax” rule.

eliminate__definition__func Replaces all function definitions with axioms.

eliminate__definition__pred Replaces all predicate definitions with axioms.

eliminate__definition Apply both transformations above.

eliminate__mutual__recursion Replaces mutually recursive definitions with axioms.

eliminate__recursion Replaces all recursive definitions with axioms.

eliminate__if__term replaces terms of the form **if formula then t2 else t3** by lifting them at the level of formulas. This may introduce **if then else** in formulas.

eliminate__if__fmla replaces formulas of the form **if f1 then f2 else f3** by an equivalent formula using implications and other connectives.

eliminate__if Apply both transformations above.

eliminate__inductive replaces inductive predicates by (incomplete) axiomatic definitions, i.e. construction axioms and an inversion axiom.

eliminate__let__fmla Eliminates **let** by substitution, at the predicate level.

eliminate__let__term Eliminates **let** by substitution, at the term level.

eliminate_let Apply both transformations above.

encoding_smt Encode polymorphic types into monomorphic type [4].

encoding_tptp Encode theories into unsorted logic.

inline_all expands all non-recursive definitions.

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

inline_trivial removes definitions of the form

```
function f x_1 .. x_n = (g e_1 .. e_k)
predicate p x_1 .. x_n = (q e_1 .. e_k)
```

when each e_i is either a ground term or one of the x_j , and each $x_1 \dots x_n$ occur at most once in the e_i

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

simplify_array Automatically rewrites the task using the lemma `Select_eq` of theory `array.Array`.

simplify_formula reduces trivial equalities $t = t$ to true and then simplifies propositional structure: removes true, false, “f and f” to “f”, etc.

simplify_recursive_definition reduces mutually recursive definitions if they are not really mutually recursive, e.g.:

```
function f : ... = .... g ...
```

```
with g : .. = e
```

becomes

```
function g : .. = e
```

```
function f : ... = .... g ...
```

if f does not occur in e

simplify_trivial_quantification simplifies quantifications of the form

```
forall x, x=t -> P(x)
```

or

```
forall x, t=x -> P(x)
```

when x does not occur in t into

```
P(t)
```

More generally, it applies this simplification whenever $x=t$ appear in a negative position.

simplify_trivial_quantification_in_goal same as above but applies only in the goal.

split_premise splits conjunctive premises.

Splitting transformations

full_split_all composition of **split_premise** and **full_split_goal**.

full_split_goal puts the goal in a conjunctive form, returns the corresponding set of subgoals. The number of subgoals generated may be exponential in the size of the initial goal.

simplify_formula_and_task same as **simplify_formula** but also removes the goal if it is equivalent to true.

split_all composition of **split_premise** and **split_goal**.

split_goal if the goal is a conjunction of goals, returns the corresponding set of subgoals. The number of subgoals generated is linear in the size of the initial goal.

split_intro when a goal is an implication, moves the antecedents into the premises.

Bibliography

- [1] A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, Edinburgh, Scotland, July 2010. Springer.
- [2] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, Berlin, Germany, Lecture Notes in Computer Science. Springer, 2007.
- [3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.
- [4] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing Polymorphism in SMT solvers. In C. Barrett and L. de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, volume 367 of *ACM International Conference Proceedings Series*, pages 1–5, 2008.
- [5] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.
- [6] S. Conchon and E. Contejean. The Alt-Ergo automatic theorem prover. <http://alt-ergo.lri.fr/>, 2008. APP deposit under the number IDDN FR 001 110026 000 S P 2010 000 1000.
- [7] L. de Moura and N. Bjørner. Z3, an efficient SMT solver. <http://research.microsoft.com/projects/z3/>.
- [8] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
- [9] B. Dutertre and L. de Moura. The Yices SMT solver. available at <http://yices.csl.sri.com/tool-paper.pdf>, 2006.
- [10] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.
- [11] G. Melquiond. Floating-point arithmetic in the Coq system. In *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 93–102, Santiago de Compostela, Spain, 2008.
- [12] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

- [13] A. Paskevich. Algebraic types and pattern matching in the logical language of the Why verification platform. Technical Report 7128, INRIA, 2009. <http://hal.inria.fr/inria-00439232/en/>.
- [14] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.
- [15] N. Shankar and P. Mueller. Verified Software: Theories, Tools and Experiments (VSTTE’10). Software Verification Competition, August 2010. <http://www.macs.hw.ac.uk/vstte10/Competition.html>.

List of Figures

1.1	The GUI when started the very first time	10
1.2	The GUI with goal G1 selected	11
1.3	The GUI after Simplify prover is run on each goal	12
1.4	The GUI after splitting goal G_2 and collapsing proved goals	12
1.5	CoqIDE on subgoal 1 of G_2	13
1.6	File reloaded after modifying goal G_2	14
2.1	Example of Why3 text.	18
2.2	Example of Why3 text (continued).	19
3.1	Solution for VSTTE'10 competition problem 1.	27
3.2	Solution for VSTTE'10 competition problem 2.	29
3.3	Solution for VSTTE'10 competition problem 3.	31
3.4	Solution for VSTTE'10 competition problem 4 (1/2).	34
3.5	Solution for VSTTE'10 competition problem 4 (2/2).	36
3.6	Solution for VSTTE'10 competition problem 5.	39
5.1	Syntax for constants.	50
5.2	Syntax for terms.	52
5.3	Syntax for formulas.	53
5.4	Syntax for theories.	55
5.5	Syntax for program types.	56
5.6	Syntax for program expressions.	57
5.7	Syntax for modules.	58

Index

- ?, 50
- _, 49, 50, 52, 55
- OB, 50
- OO, 50
- OX, 50
- Ob, 50
- Oo, 50
- Ox, 50

- 0, 50
- 1, 50
- 7, 50
- 9, 50

- A, 49, 50
- a, 49, 50
- absurd, 57
- alpha, 49
- annotation, 56
- any, 57
- as, 52, 55, 58
- assert, 57
- assert, 57
- assume, 57
- axiom, 55

- bang-op, 50
- bin-digit, 50
- binders, 53

- check, 57
- clone, 55

- decl, 55
- digit, 50
- do, 57
- done, 57
- downto, 57

- E, 50
- e, 50
- effect, 56

- Einstein's logic problem, 21
- else, 52, 53, 57
- end, 52, 53, 55, 57, 58
- exception, 58
- exists, 53
- exponent, 50
- export, 55
- expr, 57
- expr-case, 57

- F, 50
- f, 50
- false, 53
- field-value, 52, 57
- file, 54, 56
- filename, 51
- for, 57
- forall, 53
- formula, 53
- formula-case, 53
- fun, 57, 58
- function, 55
- function-decl, 55

- goal, 55

- h-exponent, 50
- hex-digit, 50
- hex-real, 50

- ident, 49
- if, 52, 53, 57
- imp-exp, 55
- import, 55, 58
- in, 52, 53, 57
- ind-case, 55
- inductive, 55
- inductive-decl, 55
- infix-op, 50
- infix-op-, 50
- integer, 50

- invariant, 57
- label, 51
- lalpha, 49
- lemma, 55
- let, 52, 53, 57, 58
- lident, 49
- logic-decl, 55
- loop, 57
- loop-annot, 57
- loop-inv, 57
- lqualid, 49
- match, 52, 53, 57
- mdecl, 58
- module, 58
- module, 58
- mrecord-field, 58
- mtime-decl, 58
- mtime-defn, 58
- mutable, 58
- namespace, 55, 58
- not, 53
- oct-digit, 50
- op-char, 50
- op-char-, 50
- P, 50
- p, 50
- pattern, 52
- pgm-decl, 58
- pgm-defn, 58
- post, 56
- post-exn, 56
- pre, 56
- predicate, 55
- predicate-decl, 55
- prefix-op, 50
- quantifier, 53
- raise, 57
- raises, 56
- raises, 56
- reads, 56
- reads, 56
- real, 50
- rec, 57, 58
- record-field, 55
- subst, 55
- subst-elt, 55
- term, 52
- term-case, 52
- then, 52, 53, 57
- theory, 55
- theory, 55
- to, 57
- to-downto, 57
- tqualid, 55
- tr-term, 53
- trigger, 53
- triggers, 53
- triple, 57
- true, 53
- try, 57
- type, 55, 58
- type, 51
- type-c, 56
- type-case, 55
- type-decl, 55
- type-defn, 55
- type-param, 55
- type-v, 56
- type-v-binder, 56
- type-v-param, 56
- ualpha, 49
- uident, 49
- uident-opt, 55
- uqualid, 49
- use, 55, 58
- val, 58
- variant, 57
- variant, 57
- while, 57
- with, 52, 53, 55, 57, 58
- writes, 56
- writes, 56
- Z, 49
- z, 49