

---

# The Why3 platform

---

Version 0.64, February 2011

François Bobot<sup>1,2</sup>  
Jean-Christophe Filliâtre<sup>1,2</sup>  
Claude Marché<sup>2,1</sup>  
Andrei Paskevich<sup>1,2</sup>

<sup>1</sup> LRI, CNRS, Univ Paris-Sud, Orsay, F-91405

<sup>2</sup> 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 of the former Why platform [9] 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** formalisations or to add support for a new external prover if wanted.

## Availability

**Why3** project page is <http://why3.gforge.inria.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 6.1 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](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, 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

<b>Contents</b>	<b>5</b>
<b>I Tutorial</b>	<b>7</b>
<b>1 Getting Started</b>	<b>9</b>
1.1 Hello Proofs . . . . .	9
1.2 Getting Started with the GUI . . . . .	9
1.3 Getting Started with the Why3 Command . . . . .	13
<b>2 The Why3 Language</b>	<b>17</b>
<b>3 The Why3 API</b>	<b>25</b>
3.1 Building Propositional Formulas . . . . .	25
3.2 Building Tasks . . . . .	26
3.3 Calling External Provers . . . . .	27
3.4 Building Terms . . . . .	29
3.5 Building Quantified Formulas . . . . .	29
3.6 Building Theories . . . . .	30
3.7 Applying transformations . . . . .	30
3.8 Writing new functions on term . . . . .	30
<b>II Reference Manual</b>	<b>31</b>
<b>4 Syntax Reference</b>	<b>33</b>
<b>5 Standard Library</b>	<b>39</b>
5.1 Library <code>bool</code> . . . . .	39
5.2 Library <code>int</code> . . . . .	39
5.3 Library <code>real</code> . . . . .	40
5.4 Library <code>floating_point</code> . . . . .	40
5.5 Library <code>array</code> . . . . .	40
5.6 Library <code>option</code> . . . . .	41
5.7 Library <code>list</code> . . . . .	41
<b>6 Reference manuals for the Why3 tools</b>	<b>43</b>
6.1 Compilation, Installation . . . . .	43
6.2 Installation of external provers . . . . .	44
6.3 The <code>why3config</code> command-line tool . . . . .	44

6.4	The <code>why3</code> command-line tool . . . . .	45
6.5	The <code>why3ide</code> GUI . . . . .	46
6.6	The <code>why3ml</code> tool . . . . .	47
6.7	The <code>why3bench</code> tool . . . . .	48
6.8	The <code>why.conf</code> configuration file . . . . .	49
6.9	Drivers of External Provers . . . . .	50
6.10	Transformations . . . . .	50
<b>Bibliography</b>		<b>53</b>
<b>List of Figures</b>		<b>55</b>
<b>Index</b>		<b>57</b>

# 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 : (false -> false) and (true or 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 6.5 for a more complete description.

The GUI is launched on the file above as follows.

```
why3ide hello_proof.why
```

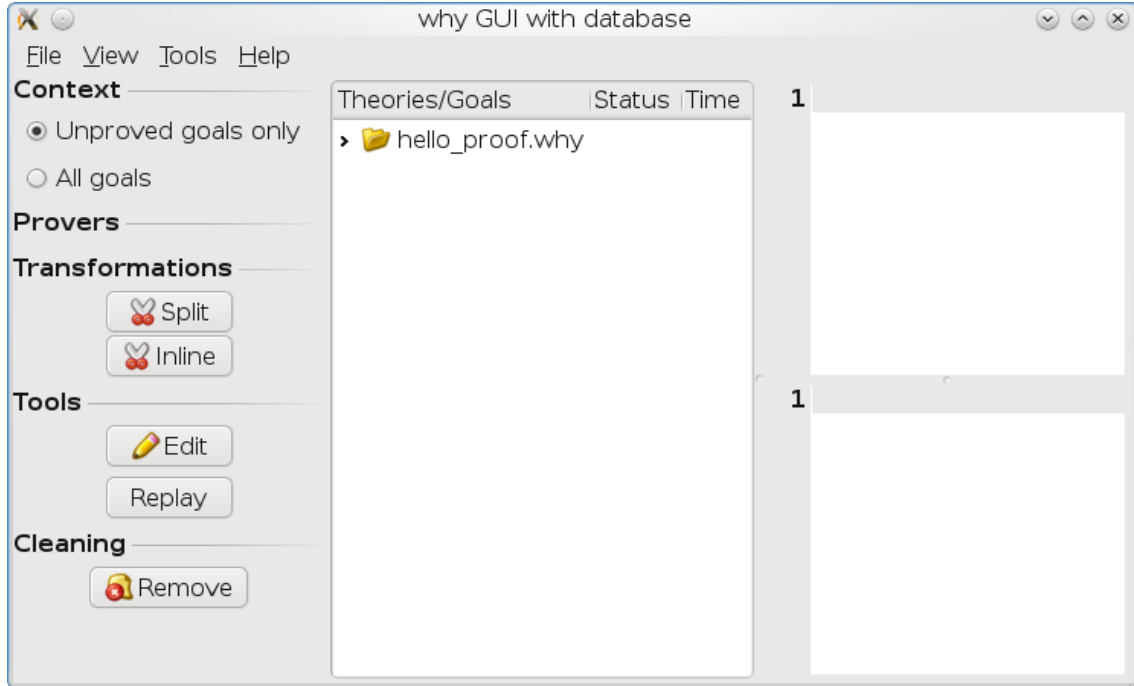


Figure 1.1: The GUI when started the very first time

When the GUI is started for the first time, you should get a window which looks like the screenshot of Figure 1.1. First of all, the left column is a tool bar which provides different actions to apply on goals. In this case, the section “Provers” is empty, which means that you have not performed prover detection yet. You should do it now using the menu **File/Detect provers**. Second, the middle part is a tree view that allows to browse inside the theories. Initially, the item of this tree are closed. We can expand this view using the menu **View/Expand all** or its shortcut **Ctrl-E**. This will result in something like the screenshot of Figure 1.2.

In the tree view, we have now a structured view of the file: this file contains one theory, itself containing three goals. In Figure 1.2, we also 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.

Notice also that three provers were detected, and are now shown in the “provers” section of the left toolbar. In this example, detected provers are Alt-Ergo [5], Coq [3] and Simplify [7].

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

The row corresponding to goal  $G_1$  is now closed, and marked with 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 are marked with an orange question

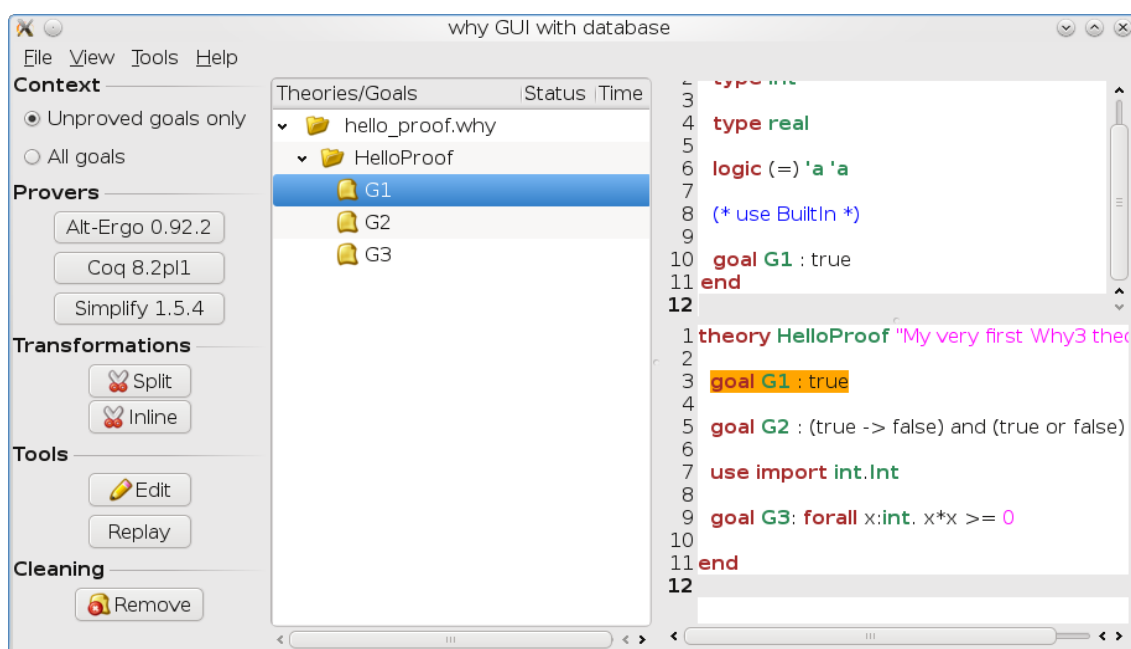


Figure 1.2: The GUI with provers detected and tree view expanded

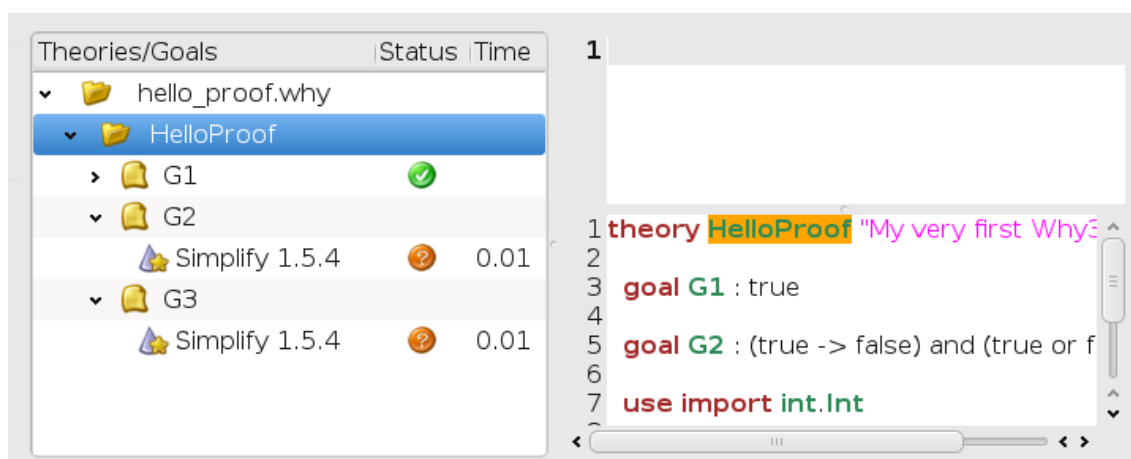
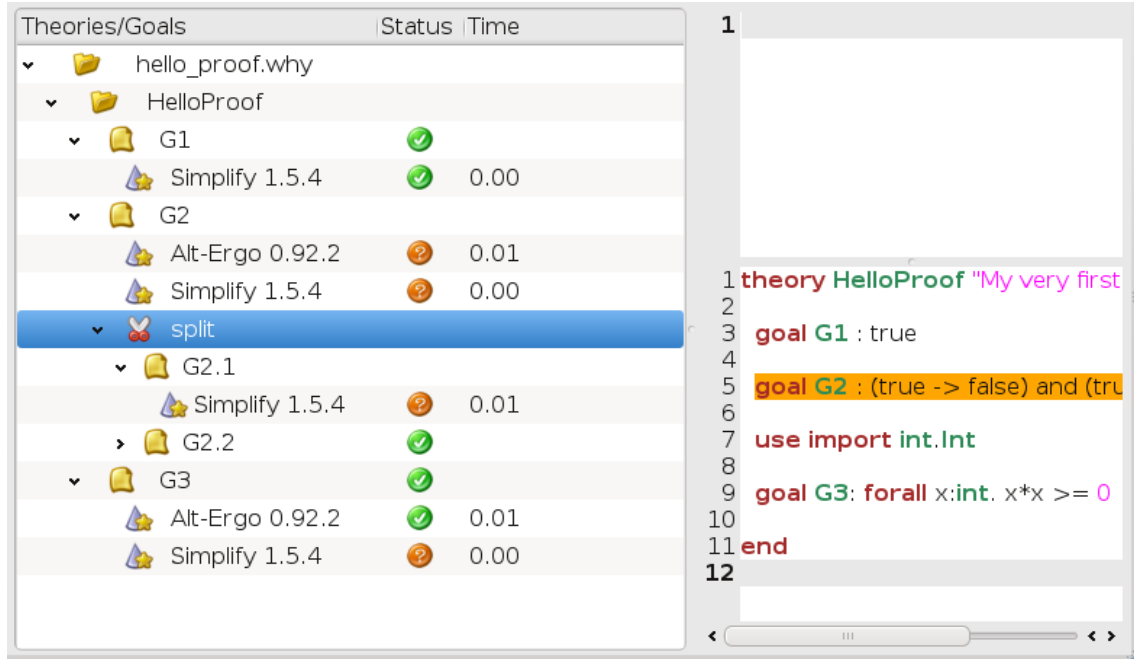


Figure 1.3: The GUI after Simplify prover is run on each goal

Figure 1.4: The GUI after splitting goal  $G_2$ 

mark.

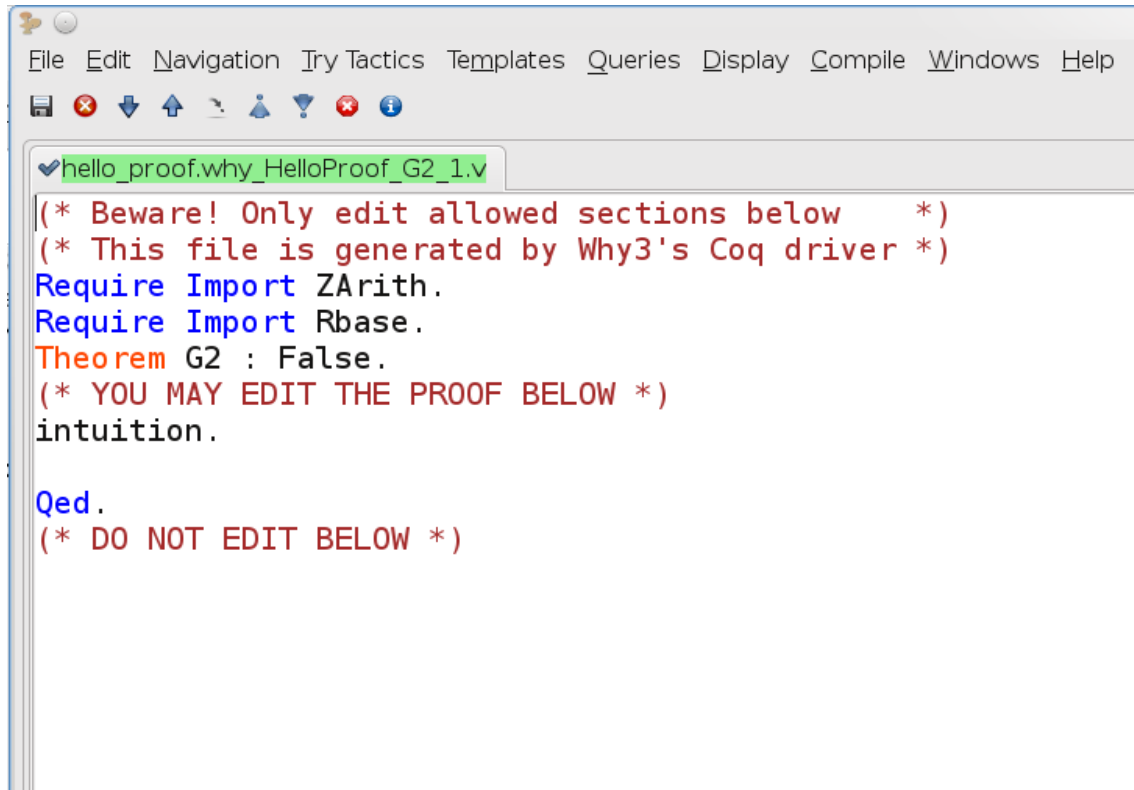
You can immediately attempt to prove the remaining goals using another prover, *e.g.* Alt-Ergo, by clicking on the corresponding button. The goal  $G_3$  should be proved now, but not  $G_2$ .

## 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. Assuming we expand everything again, 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 6.5 for details on how to configure this). You get now a regular Coq file to fill in, as 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.

Figure 1.5: CoqIDE on subgoal 1 of  $G_2$ 

### Modifying the input

Currently, the GUI does not allow to modify the input file. You must exit the GUI and modify the file 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.*

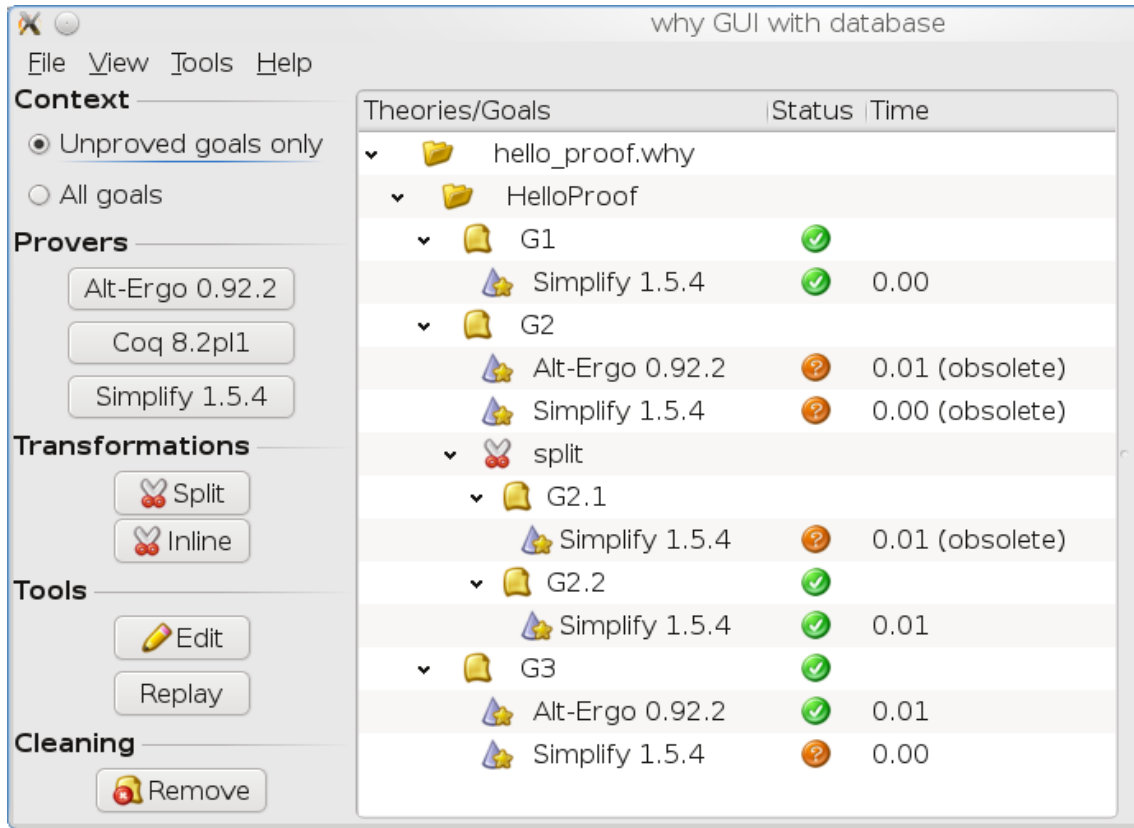
```
goal G2 : (false -> false) and (true or false)
```

Starting the IDE on the modified file and expanding everything with **Ctrl-E**, 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 SQLite3 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, using the button "Replay".

## 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 6.4 for a more complete description of this tool and all its command-line options.

Figure 1.6: The GUI restarted after modifying goal  $G_2$ 

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 [5], Coq [3] and Simplify [7].

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 Why3GUI, 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 [6.10](#) 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 a 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 `logic` keyword is used to introduce or define both function and predicate symbols. 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

  logic 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
  logic (<=) 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, logic 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 6.9).

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, logic (<=) = (<=),
     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, logic sorted1 = sorted,
     logic (<=) = (<=), 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”<sup>1</sup>. 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;

---

<sup>1</sup>This **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

  logic of t : u
  logic 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.

```

logic 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`.

```

logic rightof (h1 h2 : house) =
  leftof h2 h1
logic neighbour (h1 h2 : house) =
  leftof h1 h2 or 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 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 [TODO in Chapter rechap:apidoc.]

We assume the reader has a fair knowledge of the OCaml language. Notice that the Why3 library must be installed, see Section 6.1.

### 3.1 Building Propositional Formulas

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

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

(* a ground propositional goal: true or false *)
let fmla_true : Term.fmla = Term.f_true
let fmla_false : Term.fmla = Term.f_false
let fmla1 : Term.fmla = Term.f_or fmla_true fmla_false
```

As one can guess, the type `fmla` is the type of formulas in the library.

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

```
(* printing the formula *)
open Format
let () = printf "[formula 1 is:@ %a@]@." Pretty.print_fmla 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 or 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 logic symbols. 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.fmla = Term.f_app prop_var_A []
let atom_B : Term.fmla = Term.f_app prop_var_B []
let fmla2 : Term.fmla =
  Term.f_implies (Term.f_and atom_A atom_B) atom_A
let () = printf "@[formula 2 is:@ %a@]@." Pretty.print_fmla fmla2

```

As expected, the output is as follows.

```
formula 2 is: A and B -> A
```

Notice that the concrete syntax of Why3 forbids predicate identifiers to start with a capital letter. This constraint does not exist when building those directly using library calls.

## 3.2 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 or false
end

task 2 is:
theory Task
  logic A

  logic B

  goal Goal2 : A and B -> A
end

```

### 3.3 Calling External Provers

To call an external prover, we need to access the Why configuration file `why.conf`, as it was build 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 (Lexer.retrieve (Whyconf.loadpath main))
(* loading the Alt-Ergo driver *)
let alt_ergo_driver : Driver.driver =
  Driver.load_driver env alt_ergo.Whyconf.driver
```

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

This way to call a prover is in general too naive, since it may never return if the prover runs without time limit. The function `prove_task` has 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 `why.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 =
  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

```

### 3.4 Building Terms

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

Here is the way we build the formula  $2 + 2 = 4$ . The main difficulty is to access the internal identifier for addition: it must be retrieved from the standard theory `Int` of the file `int.why` (see Chap 5).

```

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.fmla = Term.f_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.t_app plus_symbol [two;two] Ty.ty_int

```

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

```

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

```

### 3.5 Building Quantified Formulas

To illustrate how to build quantified formulas, let us consider the formula  $\forall x : \text{int}. x * x \geq 0$ . 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.fmla = Term.f_app ge_symbol [x_times_x;zero]
```

To quantify on  $x$ , one can first build an intermediate value of type `fmla_quant`, representing a closure under a quantifier:

```
let fmla4_quant : Term.fmla_quant = Term.f_close_quant [var_x] [] fmla4_aux
let fmla4 : Term.fmla = Term.f_forall fmla4_quant
```

The second argument of `f_close_quant` is a list of triggers.

A simpler method would be to use an appropriate function:

```
let fmla4bis : Term.fmla = Term.f_forall_close [var_x] [] fmla4_aux
```

## 3.6 Building Theories

[TO BE COMPLETED]

## 3.7 Applying transformations

[TO BE COMPLETED]

## 3.8 Writing new functions on term

[TO BE COMPLETED]

# Part II

## Reference Manual





## Chapter 4

# Syntax Reference

This chapter gives the grammar for Why3 input files.

**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 `\`.

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

<i>lalpha</i>	::=	<b>a</b> - <b>z</b>   <b>_</b>
<i>ualpha</i>	::=	<b>A</b> - <b>Z</b>
<i>alpha</i>	::=	<i>lalpha</i>   <i>ualpha</i>
<i>lident</i>	::=	<i>lalpha</i> ( <i>alpha</i>   <i>digit</i>   <b>'</b> )*
<i>uident</i>	::=	<i>ualpha</i> ( <i>alpha</i>   <i>digit</i>   <b>'</b> )*
<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 4.1. 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> <sup>+</sup> <i>exponent</i>	decimal
		<i>digit</i> <sup>+</sup> . <i>digit</i> <sup>*</sup> <i>exponent</i> <sup>?</sup>	
		<i>digit</i> <sup>*</sup> . <i>digit</i> <sup>+</sup> <i>exponent</i> <sup>?</sup>	
		(0x   0X) <i>hex-real</i> <i>h-exponent</i>	hexadecimal
<i>hex-real</i>	::=	<i>hex-digit</i> <sup>+</sup>	
		<i>hex-digit</i> <sup>+</sup> . <i>hex-digit</i> <sup>*</sup>	
		<i>hex-digit</i> <sup>*</sup> . <i>hex-digit</i> <sup>+</sup>	
<i>exponent</i>	::=	(e   E) (-   +) <sup>?</sup> <i>digit</i> <sup>+</sup>	
<i>h-exponent</i>	::=	(p   P) (-   +) <sup>?</sup> <i>digit</i> <sup>+</sup>	

Figure 4.1: 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> <sup>*</sup> <i>op-char-1</i> <i>op-char</i> <sup>*</sup>
<i>infix-op</i>	::=	<i>op-char</i> <sup>+</sup>
<i>prefix-op</i>	::=	<i>op-char</i> <sup>+</sup>
<i>bang-op</i>	::=	! <i>op-char-4</i> <sup>*</sup>   ? <i>op-char-4</i> <sup>*</sup>

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*).

**Terms.** The syntax for terms is given in Figure 4.2. The various constructs have the following priorities and associativities, from lowest to greatest priority:

<i>term</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</i> [ <i>term</i> ]	brackets
		<i>term</i> [ <i>term infix-op-1 term</i> ]	ternary brackets
		<i>lqualid term</i> <sup>+</sup>	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</i> (, <i>term</i> ) <sup>*</sup> <i>with</i>	
		(  <i>term-case</i> ) <sup>+</sup> <i>end</i>	pattern matching
		( <i>term</i> (, <i>term</i> ) <sup>+</sup> )	tuple
		<i>term</i> : <i>type</i>	cast
		<i>label term</i>	label
		( <i>term</i> )	parentheses
<i>term-case</i>	::=	<i>pattern -&gt; term</i>	
<i>pattern</i>	::=	<i>pattern</i>   <i>pattern</i>	or pattern
		<i>pattern</i> , <i>pattern</i>	tuple
		-	catch-all
		<i>lident</i>	variable
		<i>uident pattern</i> <sup>*</sup>	constructor
		( <i>pattern</i> )	parentheses
		<i>pattern as lident</i>	binding

Figure 4.2: Syntax for terms.

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

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

**Type Expressions.** The syntax for type expressions notably differs from the usual ML syntax.

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

**Formulas.** The syntax for formulas is given Figure 4.3. The various constructs have the following priorities and associativities, from lowest to greatest priority:

construct	associativity
<b>if then else / let in</b>	–
label	–
<b>-&gt; / &lt;-&gt;</b>	right
<b>or /   </b>	right
<b>and / &amp;&amp;</b>	right
<b>not</b>	–
infix level 1	left
infix level 2	left
infix level 3	left
infix level 4	left
prefix	–

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

Notice that there are two symbols for the conjunction: **and** and **&&**, 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 → B*.

**Theories.** The syntax for theories is given Figure 4.4.

<i>formula</i>	<code>::=</code>	<code>true   false</code>	
		<code>formula -&gt; formula</code>	implication
		<code>formula &lt;-&gt; formula</code>	equivalence
		<code>formula and formula</code>	conjunction
		<code>formula &amp;&amp; formula</code>	asymmetric conjunction
		<code>formula or 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<sup>+</sup></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)<sup>+</sup> with</code>	
		<code>(  formula-case)<sup>+</sup> end</code>	pattern matching
		<code>quantifier binders (, binders)*</code>	
		<code>triggers<sup>?</sup> . formula</code>	quantifier
		<code>label formula</code>	label
		<code>( formula )</code>	parentheses
<i>quantifier</i>	<code>::=</code>	<code>forall   exists</code>	
<i>binders</i>	<code>::=</code>	<code>lident<sup>+</sup> : 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   formula</code>	
<i>formula-case</i>	<code>::=</code>	<code>pattern -&gt; formula</code>	

Figure 4.3: Syntax for formulas.

```

theory  ::=  theory uident label* decl* end
decl    ::=  type type-decl (with type-decl)*
          |  logic logic-decl (with logic-decl)*
          |  inductive inductive-decl (with inductive-decl)*
          |  axiom uident : formula
          |  lemma uident : formula
          |  goal uident : formula
          |  use imp-exp tqualid (as uident-opt)?
          |  clone imp-exp tqualid (as uident-opt)? subst?
          |  namespace import? uident-opt decl* end
imp-exp ::=  (import | export)?
uident-opt ::=  uident | _
subst      ::=  with (, subst-elt)+
subst-elt  ::=  type lqualid = lqualid
          |  logic lqualid = lqualid
          |  namespace (uqualid | .) = (uqualid | .)
          |  lemma uqualid
          |  goal uqualid
tqualid    ::=  uident | ident (. ident)* . uident

```

Figure 4.4: Syntax for theories.

## Chapter 5

# Standard Library

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 *)

  logic 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.

### 5.1 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`.

### 5.2 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.

### 5.3 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 `sqr` 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`.

### 5.4 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.

### 5.5 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.



## 5.6 Library option

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

## 5.7 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 6

# Reference manuals for the Why3 tools

### 6.1 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 the IDE, additional OCaml libraries are needed:

- 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>

- 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](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
```

## 6.2 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 6.3.

## 6.3 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 (`~/.why.conf` or, in the case of local installation, `why.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, 5]: <http://alt-ergo.lri.fr>

- CVC3 [2]: <http://cs.nyu.edu/acsys/cvc3/>
- Coq [3]: <http://coq.inria.fr>
- Eprover [12]: <http://www4.informatik.tu-muenchen.de/~schulz/WORK/eprover.html>
- Gappa [10]: <http://gappa.gforge.inria.fr/>
- Simplify [7]: <http://secure.ucd.ie/products/opensource/Simplify/>
- Spass : <http://www.spass-prover.org/>
- veriT : <http://www.verit-solver.org/>
- Yices [8]: <http://yices.csl.sri.com/>
- Z3 [6]: <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.

## 6.4 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 6.8.
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

## 6.5 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.  
[NOT YET AVAILABLE]

**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 [NOT YET AVAILABLE]

**Remove** Removes a proof attempt or a transformation.

## Menus

### Menu File

**Add File** adds a file in the GUI

**Detect provers** runs provers auto-detection

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

**Hide proved goals** completely hides the proved rows of the tree view [EXPERIMENTAL]

**Menu Tools** A copy of the tools already available in the left toolbar

**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.

## Structure of the database file

[TO BE COMPLETED LATER]

## 6.6 The why3ml tool

The `why3ml` is an additional layer on `Why3` library for generating verification conditions from WhyML programs. This tool and the syntax of WhyML programs is intentionally left undocumented since it might evolve significantly in the near future.

For those who want to experiment with it, examples are provided in `examples/programs`. The files `*.mlw` can be loaded in the GUI.

[TO BE COMPLETED LATER]

## 6.7 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 why.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
```

## 6.8 The why.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/.why.conf** (**\$USERPROFILE/.why.conf** under Windows) or, in the case of local installation, **why.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 = "8.2p12"

```

```
[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.

## 6.9 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]

## 6.10 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 [11]

**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\_fm1a** 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\_fm1a** 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

$$\text{logic } f \ x_1 \ \dots \ x_n = (g \ e_1 \ \dots \ e_k)$$

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.:

$$\text{logic } f : \dots = \dots g \ \dots$$

$$\text{with } g : \dots = e$$

becomes

$$\text{logic } g : \dots = e$$

$$\text{logic } f : \dots = \dots g \ \dots$$

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] S. Conchon and E. Contejean. The Alt-Ergo automatic theorem prover, 2008. <http://alt-ergo.lri.fr/>.
- [6] L. de Moura and N. Bjørner. Z3, an efficient SMT solver. <http://research.microsoft.com/projects/z3/>.
- [7] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
- [8] B. Dutertre and L. de Moura. The Yices SMT solver. available at <http://yices.csl.sri.com/tool-paper.pdf>, 2006.
- [9] 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.
- [10] 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. <http://gappa.gforge.inria.fr/>.
- [11] A. Paskevich. Algebraic types and pattern matching in the logical language of the Why verification platform. Technical Report RR-7128, INRIA, 2009. <http://hal.inria.fr/inria-00439232/en/>.
- [12] 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.



# List of Figures

1.1	The GUI when started the very first time . . . . .	10
1.2	The GUI with provers detected and tree view expanded . . . . .	11
1.3	The GUI after Simplify prover is run on each goal . . . . .	11
1.4	The GUI after splitting goal $G_2$ . . . . .	12
1.5	CoqIDE on subgoal 1 of $G_2$ . . . . .	13
1.6	The GUI restarted after modifying goal $G_2$ . . . . .	14
2.1	Example of Why3 text. . . . .	18
2.2	Example of Why3 text (continued). . . . .	19
4.1	Syntax for constants. . . . .	34
4.2	Syntax for terms. . . . .	35
4.3	Syntax for formulas. . . . .	37
4.4	Syntax for theories. . . . .	38





# Index

?, 34  
\_, 33–35, 38  
OB, 34  
O0, 34  
OX, 34  
Ob, 34  
Oo, 34  
Ox, 34  
  
0, 34  
1, 34  
7, 34  
9, 34  
  
A, 33, 34  
a, 33, 34  
alpha, 33  
and, 37  
as, 35, 38  
axiom, 38  
  
bang-op, 34  
bin-digit, 34  
binders, 37  
  
clone, 38  
  
decl, 38  
digit, 34  
  
E, 34  
e, 34  
Einstein’s logic problem, 21  
else, 35, 37  
end, 35, 37, 38  
exists, 37  
exponent, 34  
export, 38  
  
F, 34  
f, 34  
false, 37  
  
forall, 37  
formula, 37  
formula-case, 37  
  
goal, 38  
  
h-exponent, 34  
hex-digit, 34  
hex-real, 34  
  
ident, 33  
if, 35, 37  
imp-exp, 38  
import, 38  
in, 35, 37  
inductive, 38  
infix-op, 34  
infix-op-, 34  
integer, 34  
  
lalpha, 33  
lemma, 38  
let, 35, 37  
lident, 33  
logic, 38  
lqualid, 33  
  
match, 35, 37  
  
namespace, 38  
not, 37  
  
oct-digit, 34  
op-char, 34  
op-char-, 34  
or, 37  
  
P, 34  
p, 34  
pattern, 35  
prefix-op, 34  
  
quantifier, 37

*real*, 34

*subst*, 38

*subst-elt*, 38

*term*, 35

*term-case*, 35

*then*, 35, 37

*theory*, 38

*theory*, 38

*tqualid*, 38

*tr-term*, 37

*trigger*, 37

*triggers*, 37

*true*, 37

*type*, 38

*type*, 36

*ualpha*, 33

*uident*, 33

*uident-opt*, 38

*uqualid*, 33

*use*, 38

*with*, 35, 37, 38

*Z*, 33

*z*, 33